

# ***Dreamcast (VMU) Visual Memory Unit***

***Tutorial Manual  
Specifications  
Hardware Manual  
Programing Manual  
VMU-BIOS Specifications  
Sound Development Specifications  
Simulator Manual***



# Table of Contents

**Visual Memory Unit (VMU) Tutorial Manual . . . . . VMT-i**

**Table of Contents . . . . . VMT-iii**

**Application Development Procedure . . . . . VMT-1**

Writing Source Code . . . . . VMT-1  
 Correcting GHEAD.ASM . . . . . VMT-2  
 Assembly Without Using MAKE . . . . . VMT-2  
     Assembly . . . . . VMT-3  
     Linking . . . . . VMT-3  
     Converting an EVA File Into a HEX File . . . . . VMT-4  
     Converting a HEX File to a Binary File . . . . . VMT-5  
     Creating a MAKE File . . . . . VMT-5  
 Creating the Information Fork . . . . . VMT-7  
 Transferring the Program to Visual Memory . . . . . VMT-7

**Interfacing between Visual Memory and Dreamcast . . . . . VMT-9**

Names of Elements in the Startup Screen . . . . . VMT-10  
     Memory Selection Screen . . . . . VMT-10  
     File Management Screen . . . . . VMT-11  
 Creating a Volume Icon . . . . . VMT-13  
 Creating an Animated Icon . . . . . VMT-15  
     Three File Structures . . . . . VMT-15  
     Information Fork . . . . . VMT-16  
     Visual Comment Data Structure . . . . . VMT-20  
     Game Name Sorting Rules . . . . . VMT-21

**Memory Card Utility..... VMT-23**

Memory Card Utility Preparation and Startup ..... VMT-23  
    Requirements for Transfer ..... VMT-23  
    Software Preparation ..... VMT-24  
    Memory Card Utility Startup ..... VMT-26  
Memory Card Utility Operation ..... VMT-27  
    Main Menu ..... VMT-27  
    Memory Selection Menu ..... VMT-27  
    Command Selection Menu ..... VMT-28  
    File Operations Menu ..... VMT-31  
Initializing Visual Memory ..... VMT-33  
Transferring Files from a PC to Visual Memory ..... VMT-34  
LCD Pattern Display ..... VMT-40  
LCD Character Pattern Display ..... VMT-44  
Counter That Uses Base Timer Interrupts ..... VMT-50  
Button Press Detection ..... VMT-58  
Using the PWM Sound Source ..... VMT-64  
Interrupt Using Timer 0 ..... VMT-66  
Serial Communications (Sending Side) ..... VMT-72  
Serial Communications (Receiving Side) ..... VMT-80  
General-purpose Serial Driver ..... VMT-88  
Reading and Writing Flash Memory ..... VMT-102  
Low Battery Detection and Saving Data ..... VMT-111

**Dreamcast VMU Specifications ..... VMU-i**

**Table of Contents ..... VMU-iii**

**VMU Specifications..... VMU-1**

Overview ..... VMU-1  
    VMU Overview ..... VMU-1  
    VMU Configuration ..... VMU-2  
    VMU Functions ..... VMU-4  
Mode Settings ..... VMU-7  
File Management ..... VMU-9  
    Management Area ..... VMU-10  
    Data Area ..... VMU-10  
    Reserved Area ..... VMU-10  
LCD Display ..... VMU-11  
    XRAM ..... VMU-11  
    Screen Mode ..... VMU-11  
    Icons ..... VMU-12  
    Screen Configuration ..... VMU-12  
    LCD Characteristics ..... VMU-12  
    Miscellaneous ..... VMU-12

Executable File Initiation .....	VMU-13
Downloading an Executable File .....	VMU-13
File Size .....	VMU-13
Subroutine .....	VMU-13
Interrupts .....	VMU-14
RAM .....	VMU-14
Save Processing During Executable File Operations .....	VMU-14
Auto Power Off .....	VMU-14
Communications Function .....	VMU-15
Maple Bus Protocol .....	VMU-15
Synchronous Serial Communications .....	VMU-15
Clock Function .....	VMU-16
Settings .....	VMU-16
Alarm Function .....	VMU-17
SLEEP Function .....	VMU-18
SLEEP Operation .....	VMU-18
Buttons .....	VMU-19
Batteries .....	VMU-20
Battery Life .....	VMU-20
Processing When Battery Power Is Exhausted .....	VMU-20
Battery Replacement .....	VMU-20
Postscript .....	VMU-20

## **Visual Memory Unit (VMU) Hardware Manual . . . . . VMD-i**

### **Table of Contents . . . . . VMD-iii**

### **Visual Memory Unit Overview . . . . . VMD-1**

VMU Specifications .....	VMD-2
VMU Functions .....	VMD-6
File management .....	VMD-7
Liquid-Crystal Display .....	VMD-7
Starting VMU applications .....	VMD-7
Data transfer .....	VMD-7
Clock .....	VMD-7
Buzzer .....	VMD-8
Operation mode switching .....	VMD-8
Integrated character font .....	VMD-8
Mode Setting .....	VMD-9
System mode .....	VMD-9
Game mode .....	VMD-9
File mode .....	VMD-10
Clock mode .....	VMD-10
File Management .....	VMD-11
Flash memory management area .....	VMD-11
Data area .....	VMD-13
Reserved area .....	VMD-13

LCD Display .....	VMD-14
XRAM .....	VMD-14
Image mode .....	VMD-14
Icon .....	VMD-14
Image configuration .....	VMD-14
LCD characteristics .....	VMD-15
Other important points .....	VMD-15
Starting an Executable File .....	VMD-16
Writing applications for the VMU .....	VMD-16
Transferring an executable file .....	VMD-16
Executable file size .....	VMD-16
OS programs usable by applications .....	VMD-16
RAM .....	VMD-17
Saving application data .....	VMD-17
Auto power-off .....	VMD-18
Communication Functions .....	VMD-19
Maple bus protocol .....	VMD-19
Synchronous serial transfer .....	VMD-19
Clock Function .....	VMD-20
Alarm Function .....	VMD-21
Sleep Function .....	VMD-22
Buttons .....	VMD-23
Batteries .....	VMD-24
Battery life .....	VMD-25
Battery status monitoring .....	VMD-25
Battery replacement .....	VMD-25

## **CPU Features . . . . . VMD-27**

Differences to Conventional CPUs .....	VMD-28
Specifications .....	VMD-29
System block diagram .....	VMD-33

## **Internal System Configuration . . . . . VMD-35**

Memory Space .....	VMD-35
Program Counter (PC) .....	VMD-36
ROM Space .....	VMD-38
RAM Space .....	VMD-38
Indirect Address Registers .....	VMD-39
Special function registers (SFR) .....	VMD-40
Flash Memory .....	VMD-43
Accumulator .....	VMD-43
B Register, C Register .....	VMD-43
Program Status Word (PSW) .....	VMD-44
Stack Pointer .....	VMD-46
Table Reference Register (TRR) .....	VMD-47
CHANGE Instruction .....	VMD-48
Format .....	VMD-48
Operation .....	VMD-48
Sample program .....	VMD-48

# Peripheral System Configuration . . . . . VMD-49

I/O Ports .....	VMD-49
Port 1 .....	VMD-50
Port 3 .....	VMD-54
Port 7 .....	VMD-56
Timer/Counter 0 (T0) .....	VMD-58
Functions .....	VMD-58
Circuit Configuration .....	VMD-59
Related Registers .....	VMD-60
Circuit Configuration and Operation Principles .....	VMD-69
Timer 1 (T1) .....	VMD-76
Functions .....	VMD-76
Circuit Configuration .....	VMD-77
Related Registers .....	VMD-78
Circuit Configuration and Operation Principles .....	VMD-82
Base Timer .....	VMD-94
Functions .....	VMD-94
Circuit Configuration .....	VMD-95
Related Registers .....	VMD-96
Using the Base Timer .....	VMD-99
Serial Interface .....	VMD-100
Functions and Features .....	VMD-100
Circuit Configuration .....	VMD-102
Related Registers .....	VMD-103
Serial Interface Operation .....	VMD-109
Operation Mode Settings .....	VMD-109
Serial transfer clock .....	VMD-111
Serial Transfer Timing .....	VMD-113
LSB/MSB Switchable Start Sequence .....	VMD-114
Overrun Detection .....	VMD-116
Transfer Bit Length Control .....	VMD-117
Sample Program .....	VMD-117
Dot Matrix LCD Controller .....	VMD-120
Functions .....	VMD-120
Display RAM (XRAM) .....	VMD-120
Display Control Registers .....	VMD-121
External Interrupt Function .....	VMD-128
Circuit Configuration .....	VMD-129
Related Registers .....	VMD-129
Port Interrupt Functions .....	VMD-135
Function .....	VMD-135
Circuit Configuration .....	VMD-135
Related Registers .....	VMD-136
Operation Description .....	VMD-137
State Transition .....	VMD-137
VMU Work RAM .....	VMD-139
Work RAM Control Registers .....	VMD-139
Accessing Work RAM .....	VMD-140
Precautions for Using Work RAM Address Register .....	VMD-140
Flash Memory .....	VMD-142
Features and Functions .....	VMD-142
Accessing Program/Data Area of Flash Memory .....	VMD-142

**Control Functions . . . . . VMD-143**

Interrupt Functions ..... VMD-143  
    Interrupt Types ..... VMD-144  
    Interrupt Function Operation ..... VMD-145  
    Circuit Configuration ..... VMD-146  
    Related Registers ..... VMD-147  
    Interrupt Priority Ranking ..... VMD-150  
System Clock Generation ..... VMD-151  
    Features and Functions ..... VMD-153  
    Circuit Configuration ..... VMD-154  
    Related Registers ..... VMD-156  
    System Clock Operation Mode ..... VMD-159  
Sleep Function ..... VMD-161  
    Related Registers ..... VMD-162  
    Standby Operation Status ..... VMD-163  
    HALT Mode ..... VMD-164  
Hardware Reset Function ..... VMD-165  
    External Reset Pin Function ..... VMD-166  
    Hardware Status During a Reset ..... VMD-167

**Programs in ROM . . . . . VMD-171**

System Programs ..... VMD-172  
OS Programs ..... VMD-173  
Headers ..... VMD-174

**Memory Space . . . . . VMD-175**

**System BIOS Functions . . . . . VMD-177**

**Subroutine Call Procedure . . . . . VMD-179**

Processing Contents of Labels ..... VMD-180  
Interaction Between System BIOS and Application ..... VMD-181

**Application Shutdown Procedure When MODE Button is Pressed . . . . . VMD-183**

Processing Contents of Labels ..... VMD-184  
Interaction Between System BIOS and Application ..... VMD-185

**VMU Initialization . . . . . VMD-187**



<b>Subroutine Reference</b> .....	<b>VMD-189</b>
Flash Memory Access Functions .....	VMD-189
Subroutine Use Precautions .....	VMD-190
Flash memory routines .....	VMD-192
fm_prd_ex(ORG 0120H) Flash memory page data read .....	VMD-192
fm_wrt_ex(ORG 0100H) Flash memory data write .....	VMD-194
fm_vrf_ex(ORG 0110H) Flash memory page data verify .....	VMD-195
Clock Function .....	VMD-198
timer_ex Clock count-up timer .....	VMD-198
 <b>Low Battery Voltage Auto Detection</b> .....	 <b>VMD-199</b>
 <b>List of Defined Variables</b> .....	 <b>VMD-201</b>
 <b>Sound Output Method</b> .....	 <b>VMD-203</b>
Timer 1 Outline .....	VMD-203
Timer 1 Block Configuration .....	VMD-203
Related Registers .....	VMD-204
Mode Setting .....	VMD-205
8 Bit Counter Mode .....	VMD-206
Output Waveform and Parameter Setting .....	VMD-206
8 Bit Counter Mode Setting .....	VMD-207
Frequency Characteristics .....	VMD-208
Output Frequency Table .....	VMD-208
 <b>Sample Program</b> .....	 <b>VMD-211</b>
 <b>Variable Bit Length Pulse Generator</b> .....	 <b>VMD-213</b>
 <b>Symbol Table</b> .....	 <b>VMD-217</b>
 <b>VMU Mode Selection</b> .....	 <b>VMD-221</b>
 <b>Calculation of Battery Life</b> .....	 <b>VMD-223</b>
Methods for Enhancing Battery Life .....	VMD-223
Oscillator Circuit and Current Consumption .....	VMD-224
Oscillation Control Register .....	VMD-224
System Clock Division Ratio Setting .....	VMD-224
Oscillator Circuit Selection .....	VMD-224
Oscillator Circuit Start/Stop .....	VMD-225
Calculating Battery Life .....	VMD-225
Calculating Continuous Operating Time .....	VMD-225
Calculating Battery Life in Days .....	VMD-226
 <b>Serial Communication Precautions</b> .....	 <b>VMD-229</b>
Serial Communication Timing Chart .....	VMD-229
Measures to Ensure Problem-Free Serial Transfer .....	VMD-230
Mask All Interrupts .....	VMD-230
Set Maximum Send Wait Time .....	VMD-231

# **Visual Memory Unit (VMU) Programing Manual . . . . . VMC-i**

## **Table of Contents . . . . . VMC-iii**

### **Setup. . . . . VMC-1**

Executing the Setup Program . . . . . VMC-1

Post-Installation Overview . . . . . VMC-7

### **Setting Environment Variables . . . . . VMC-9**

Environment Variables for the Development Tools . . . . . VMC-9

    Environment Variable Settings . . . . . VMC-10

### **Specifying Files for Assembly. . . . . VMC-11**

Specifying File Names . . . . . VMC-11

Specifying Parameters on the Command Line . . . . . VMC-12

Specifying Parameters at the Prompts . . . . . VMC-13

### **Option Switches. . . . . VMC-15**

### **Environment Variables and Reserved Word File . . . . . VMC-17**

Environment Variables . . . . . VMC-18

Reserved Word File . . . . . VMC-19

### **Errors . . . . . VMC-21**

Warnings . . . . . VMC-22

Non-Fatal Errors . . . . . VMC-25

Fatal Errors . . . . . VMC-31

### **Listing Format . . . . . VMC-35**

### **Specifying Files for Linking . . . . . VMC-39**

Specifying File Names . . . . . VMC-40

Specifying Parameters on the Command Line . . . . . VMC-41

Specifying Parameters at the Prompts . . . . . VMC-42

Files Referenced During Linking . . . . . VMC-44

### **Option Switches. . . . . VMC-45**

**Object Alignment . . . . . VMC-49**

-A option ..... VMC-50  
-A -F options ..... VMC-51  
-A -O options ..... VMC-52  
-A -R options ..... VMC-53

**Errors . . . . . VMC-55**

Fatal Errors ..... VMC-55  
Non-Fatal Errors ..... VMC-56

**Starting the Program . . . . . VMC-57**

Specifying File Names ..... VMC-57  
Specifying Parameters on the Command Line ..... VMC-58  
    Option ..... VMC-59  
    Examples of Command Line Execution ..... VMC-59  
Operation with the Prompts ..... VMC-60  
    Prompt Line Extension ..... VMC-60  
    Default Responses ..... VMC-60

**Error Messages . . . . . VMC-61**

**Cross-Reference . . . . . VMC-63**

**Starting the Program . . . . . VMC-65**

Specifying File Names ..... VMC-66  
Specifying Parameters ..... VMC-67  
Option Specification ..... VMC-68

**Error Messages . . . . . VMC-69**

Fatal Errors ..... VMC-69

**Starting the Program . . . . . VMC-71**

Specifying File Names ..... VMC-71  
Specifying Parameters ..... VMC-72

**Error Messages . . . . . VMC-73**

Fatal Errors ..... VMC-73

<b>Overview of MAKE</b> .....	<b>VMC-75</b>
Running MAKE .....	VMC-76
Build Priority Sequence .....	VMC-76
Command Line Options .....	VMC-76
Makefile Syntax .....	VMC-78
Generation Rules .....	VMC-78
Macros .....	VMC-80
Directives .....	VMC-81
Implicit Rules .....	VMC-82
Makerule file .....	VMC-82

<b>Assembler Syntax</b> .....	<b>VMC-85</b>
Statements .....	VMC-85
Label and Symbol Names .....	VMC-86
Comments .....	VMC-86
Operators .....	VMC-86
Numeric Constants .....	VMC-87
Character Constants .....	VMC-88
Character String Constants .....	VMC-89
Special Symbols .....	VMC-89

<b>Assembler Pseudoinstructions</b> .....	<b>VMC-91</b>
---	---------------

<b>LC86K Instruction Summary</b> .....	<b>VMC-147</b>
Instruction Summary .....	VMC-147
Arithmetic Instructions .....	VMC-147
Logical Instructions .....	VMC-148
Data Transfer Instructions .....	VMC-148
Jump Instruction .....	VMC-148
Conditional Branch Instructions .....	VMC-149
Subroutine Instruction .....	VMC-149
Bit Manipulation Instructions .....	VMC-149
Other Instructions .....	VMC-149
Macro Instruction .....	VMC-149
Addressing .....	VMC-149
Program Memory Addressing .....	VMC-150
RAM and Special Function Register (SFR) Addressing .....	VMC-152

<b>Instruction Set Reference</b> .....	<b>VMC-155</b>
Arithmetic Instructions .....	VMC-156
Logical Instructions .....	VMC-173
Data Transfer Instructions .....	VMC-186
Jump Instructions .....	VMC-197
Conditional Branch Instructions .....	VMC-201
Subroutine Instructions .....	VMC-214
Bit Manipulation Instructions .....	VMC-219
Miscellaneous Instruction .....	VMC-222
Macro Instruction .....	VMC-223

**LC86K Instruction Set Summary . . . . . VMC-225**

**Assembler Pseudoinstructions . . . . . VMC-227**

**Visual Memory Unit (VMU) VMU-BIOS Specifications . . . . . VME-i**

**VMU-BIOS Specifications . . . . . VME-1**

Outline ..... VME-1  
VMU Outline ..... VME-2  
    System-BIOS Outline ..... VME-2  
Memory Space ..... VME-3  
System BIOS Functions ..... VME-5  
System BIOS Data and Memory Allocation ..... VME-6  
    Program Layout ..... VME-6  
    Subroutine Call Flow ..... VME-7  
    Returning From User Program to Mode Selection Screen ..... VME-9  
    VMU Initialization ..... VME-10  
Subroutine Description ..... VME-12  
    Flash Memory Access Functions ..... VME-12  
    Clock Function ..... VME-19  
Automatic low battery detection function ..... VME-20  
    Automatic low battery detection flag ..... VME-20

**Visual Memory Unit (VMU)  
Sound Development Specifications . . . . . VMA-i**

**Table of Contents . . . . . VMA-iii**

**VMU Sound Development Specifications . . . . . VMA-1**

VMU Sound Output Hardware Outline ..... VMA-1  
Sound Output Principle ..... VMA-2  
    Timer 1 Outline ..... VMA-2  
    8-Bit Counter Mode ..... VMA-5  
    Table of Available Output Frequencies ..... VMA-8  
Sample Program ..... VMA-13

**Visual Memory Unit (VMU) Simulator Manual . . . . . VMB-i**

**Table of Contents . . . . . VMB-iii**

**Overview . . . . . VMB-1**

Features ..... VMB-1  
Visual Memory Simulator Operating Environment ..... VMB-2  
Checking Operation on Actual Visual Memory Hardware ..... VMB-3  
Notes Concerning Startup for the First Time ..... VMB-4

## **Implemented Devices ..... VMB-5**

Virtual CPU .....	VMB-5
Memory .....	VMB-6
LCD Controller (LCDC) .....	VMB-6
Serial Interface (SIO) .....	VMB-7
Timer .....	VMB-7
Interrupt Controller .....	VMB-7
I/O Ports .....	VMB-7
External Input Devices .....	VMB-8

## **Basic Operation ..... VMB-9**

Starting Up the Visual Memory Simulator .....	VMB-9
Loading the System BIOS .....	VMB-10
Loading and Executing Applications .....	VMB-11
MAP File .....	VMB-12
Drag & Drop .....	VMB-12

## **Descriptions of Windows and Panels ..... VMB-13**

Main Window .....	VMB-14
Menus .....	VMB-14
Toolbar .....	VMB-17
CPU Register Display Function .....	VMB-18
Execution Control .....	VMB-19
Disassembly Function .....	VMB-20
Visual Memory Image .....	VMB-21
Status Lamp .....	VMB-21
Changing the Size of the Main Window .....	VMB-22
System Console .....	VMB-22
Memory Control Window .....	VMB-23
RAM#0, RAM#1 .....	VMB-24
FLASH#0 .....	VMB-25
XRAM .....	VMB-26
SFR .....	VMB-27
VTRBF .....	VMB-28
Break Control Window .....	VMB-29
Break by Breakpoint Address Comparison .....	VMB-29
Display When an Interrupt Is Received .....	VMB-32
Access Reference Monitor .....	VMB-33
Special Function Register Control Window .....	VMB-34
CPU Control .....	VMB-35
LCD .....	VMB-35
INT Control .....	VMB-36
Timer 0 .....	VMB-36
Timer 1 .....	VMB-37
SIO .....	VMB-37
PORT1 .....	VMB-38
PORT3/7 .....	VMB-38
External INT .....	VMB-39
VMU Special .....	VMB-40
Base Timer .....	VMB-40

LCD Snapshot Window .....	VMB-41
Description of Tool Bar Buttons .....	VMB-41
Display by STAD Checkbox .....	VMB-42
Menus .....	VMB-42
Network Monitor Window .....	VMB-43
Trace Panel .....	VMB-45
Hexadecimal Input Pad .....	VMB-47
Environment Settings Window .....	VMB-49
Settings .....	VMB-49
Work Settings .....	VMB-51

**Networking .....** VMB-55

**Related Files .....** VMB-57

System Files .....	VMB-58
Application Files .....	VMB-59

**Warning Messages .....** VMB-61





***Visual Memory Unit (VMU)***  
***Tutorial Manual***



# ***Table of Contents***

<b>Application Development Procedure .....</b>	<b>VMT-1</b>
Writing Source Code .....	VMT-1
Correcting GHEAD.ASM .....	VMT-2
Assembly Without Using MAKE .....	VMT-2
Assembly .....	VMT-3
Linking .....	VMT-3
Converting an EVA File Into a HEX File .....	VMT-4
Converting a HEX File to a Binary File .....	VMT-5
Creating a MAKE File .....	VMT-5
Creating the Information Fork .....	VMT-7
Transferring the Program to Visual Memory .....	VMT-7
 <b>Interfacing between Visual Memory and Dreamcast .....</b>	 <b>VMT-9</b>
Names of Elements in the Startup Screen .....	VMT-10
Memory Selection Screen .....	VMT-10
File Management Screen .....	VMT-11
Creating a Volume Icon .....	VMT-13
Creating an Animated Icon .....	VMT-15
Three File Structures .....	VMT-15
Information Fork .....	VMT-16
Visual Comment Data Structure .....	VMT-20
Game Name Sorting Rules .....	VMT-21

## **Memory Card Utility..... VMT-23**

Memory Card Utility Preparation and Startup .....	VMT-23
Requirements for Transfer .....	VMT-23
Software Preparation .....	VMT-24
Memory Card Utility Startup .....	VMT-26
Memory Card Utility Operation .....	VMT-27
Main Menu .....	VMT-27
Memory Selection Menu .....	VMT-27
Command Selection Menu .....	VMT-28
File Operations Menu .....	VMT-31
Initializing Visual Memory .....	VMT-33
Transferring Files from a PC to Visual Memory .....	VMT-34
LCD Pattern Display .....	VMT-40
LCD Character Pattern Display .....	VMT-44
Counter That Uses Base Timer Interrupts .....	VMT-50
Button Press Detection .....	VMT-58
Using the PWM Sound Source .....	VMT-64
Interrupt Using Timer 0 .....	VMT-66
Serial Communications (Sending Side) .....	VMT-72
Serial Communications (Receiving Side) .....	VMT-80
General-purpose Serial Driver .....	VMT-88
Reading and Writing Flash Memory .....	VMT-102
Low Battery Detection and Saving Data .....	VMT-111

# Application Development Procedure

---

This chapter explains the application development procedure, from coding the program to checking the program on an actual machine. This section assumes that the application specifications have already been established.

## Writing Source Code

The following declarations must be made at the start of the program:

```
chip LC868700
World external
Public main
Extern _game_end
```

Because all Visual Memory applications will be stored in flash memory, "external" must be declared in the "world" statement.

When an application is called from system BIOS, address 0000H in flash memory is called. Because "jump main" is written in 0000H by GHEAD.ASM, the application provides the label "main" for entry into game mode. Because "main" is referenced from GHEAD.ASM, the "public" declaration is used.

Conversely, when an application ends, it jumps to "\_game\_end" in GHEAD.ASM, so the "extern" declaration is used to indicate that this label is external to the application.

When an application calls a flash memory-related BIOS or a clock-related BIOS, the "extern" declaration is used to indicate that "fm\_wrt\_ex", "fm\_vrf\_ex", "fm\_prd\_ex", etc., are external programs.

Next, the structure of the indirect address register for the data segment (DSEG) is defined. The entire data segment is expanded in RAM. Because addresses 0000 to 000FH in RAM are indirect address registers, 16 bytes of RAM should be allocated for these registers, whether or not the application will use the indirect address registers. The area in RAM that can be used by an application starts from address 0010H.

---

**Reference:** For details on indirect address registers, refer to the "Visual Memory Hardware Manual."

---

Start the code segment (CSEG) from an address higher than 0280H (org 280H). GHEAD . ASM uses 0000H to 01FFH, and the information fork uses 0200H to 027FH (minimum).

GHEAD . ASM contains interrupt vector definitions, and BIOS cal and return destinations. The information fork data such as the application name and icon. The size of the information fork is variable because of choices that the designer can make: the icon may or may not be animated, or one application's icon may be larger than another's.

If the information fork image is pre-determined, add the size of the information fork image to 0200H and start the program from that address.

---

**Reference:** For details on the information fork, refer to Chapter 2, "Interfacing between Visual Memory and Dreamcast."

---

## Correcting GHEAD.ASM

Once you have written the application source code, it is necessary to correct GHEAD . ASM.

If the application uses interrupts, describe the vector table for the interrupts that are to be used in GHEAD . ASM.

Even if interrupts are not to be used, it is still necessary to define an interrupt vector table. Also write the interrupt handler so that it does nothing except execute "RETI".

Because the program jumps to the start (0000H) of GHEAD . ASM if the user selects game mode, a jump instruction to the main routine of the game should be written at the start of GHEAD . ASM.

The processing that is to be performed for BIOS calls is described starting from address 100H. Do not change this processing. Because the BIOS in ROM specifies addresses directly and then returns control to flash memory, BIOS calls will not be made correctly if addresses change by even one byte.

When writing to flash memory in particular, it is necessary to use 1/6 RC for the system clock. Make this change within the application program by calling `fm_wrt_ex`, `fm_vrf_ex`, `fm_prd_ex`, and then changing over to crystal oscillation when control returns to the main program.

---

**Note:** When loading from flash memory, it does not matter if 1/6 or 1/12 RC is used.

---

Also be careful not to change the "org" instruction that specifies each BIOS start address.

## Assembly Without Using MAKE

This section explains how to assemble and link the source code, and then build a file in a format that can be actually executed in Visual Memory.

---

**Caution:** The Assembler and Linker use EMS. Before starting, display the MS-DOS prompt properties and enable EMS memory usage in the "EMS Memory" group under the "Memory" tab. EMS memory cannot be used when `EMM386 . EXE` is embedded in `CONFIG . SYS` and the `NOEMS` option is specified, so the `NOEMS` option must be removed.

---

### Assembly

Execute the "M86K" command from the MS-DOS command prompt to assemble the source code.

For example, if the source code file name is "TEST.ASM", set the current drive and the current directory to the directory where "TEST.ASM" resides, and then perform the assembly process by executing the following command:

```
C>M86K TEST.ASM
SANYO (R) LC86K series Macro Assembler Version 4.0K
Copyright (c) SANYO Electric Co., Ltd. 1989-1995. All rights reserved.

Pass 1 .....
Source file:   TEST
Chip name:    LC868700
ROM size:    60K bytes
RAM size:    512 bytes
XRAM size:   196 bytes
Pass 2 .....
```

When assembly is completed, an object file with the extension ".OBJ" is created.

Assembling GHEAD.ASM in the same manner creates GHEAD.OBJ.

If an error message similar to the following appears during the assembly process, a problem exists in the line indicated by the line number in the message.

```
Pass 1 .....
TEST.ASM(93):          move      #080h,b
** Error, syntax error near #
0 warning(s) and 1 error(s) were detected. Further execution aborted.
```

Correct the source code so that no warnings or errors are generated.

---

**Reference:** For details on the M86K assembler's warning messages and error messages, refer to "Visual Memory Programmer's Manual."

---

**Caution:** Except when incorporating source code equivalent to GHEAD.ASM into your own source code, GHEAD.OBJ and the user program object file are both required. Note that interrupt vectors, interrupt service routines, BIOS call programs, etc., are described in GHEAD.ASM, and are placed in addresses below the user program by the Linker that is executed next.

---

### Linking

After preparing an object file (created by the Assembler) and a GDUMMY.OBJ file that indicates the addresses in internal ROM where BIOS is written, use the Linker to create an EVA-format file.

---

**Note:** An EVA-format file is a file that uses special debugging hardware. Because the Visual Memory Simulator is used in the development of applications for Visual Memory, think of the EVA file as a temporary file.

---

## Visual Memory Unit (VMU) Tutorial Revision

---

Before executing the linker, make a note of the GDUMMY.OBJ path. Then input the following command line to link each of the object files.

```
D>L86K GHEAD.OBJ IFORK.OBJ TEST.OBJ -C=200 C:\VM_SDK\LC86K\OBJ\GDUMMY.OBJ,
TEST.EVA,,,
SANYO (R) LC86K series Linkage Loader Version 6.00c
Copyright (c) SANYO Electric Co., Ltd. 1989-1997. All right reserved.
Pass 1 ...
Pass 2 ...
Pass 3 ...
Link process complete !!
TEST.EVA created
```

The option "-C=200" specifies the address in flash memory where the user program that is specified immediately afterwards is to be placed.

If an error message is displayed, review GHEAD.ASM and the user program. Check the labels that are used for BIOS calls in particular.

## Converting an EVA File Into a HEX File

The Linker combines all of the object files into a single file with the ".EVA" extension. The next step is to convert this file into a file that can be loaded into Visual Memory or the Visual Memory Simulator. Input the following command line.

```
D>E2H86K TEST.EVA
SANYO LC86000 Series EVA-file to HEX-file generator V1.21A
Copyright (C) SANYO Electric Co.,Ltd. 1992-1997
EVA file name:          TEST.EVA
ROM data packed:       FF(hex)
Chip name:             LC868716
All ROM(64KB) block    records: 03875
All ROM(64KB) block    records: 04096
Module name: GHEAD     External CSEG(In)    0000 - 0002    records: 00001
Module name:           External CSEG(In)    0003 - 0004    records: 00001
Module name:           External CSEG(In)    000B - 000C    records: 00001
Module name:           External CSEG(In)    0013 - 0014    records: 00001
Module name:           External CSEG(In)    001B - 001C    records: 00001
Module name:           External CSEG(In)    0023 - 0024    records: 00001
Module name:           External CSEG(In)    002B - 002C    records: 00001
Module name:           External CSEG(In)    0033 - 0034    records: 00001
Module name:           External CSEG(In)    003B - 003C    records: 00001
Module name:           External CSEG(In)    0043 - 0044    records: 00001
Module name:           External CSEG(In)    004B - 0057    records: 00002
Module name:           External CSEG(In)    0100 - 0105    records: 00001
Module name:           External CSEG(In)    0110 - 0115    records: 00001
Module name:           External CSEG(In)    0120 - 0125    records: 00001
Module name:           External CSEG(In)    0130 - 013B    records: 00001
Module name:           External CSEG(In)    01F0 - 01F4    records: 00001
Module name: IFORK     External CSEG(In)    01F5 - 0474    records: 00041
Module name: TEST      External CSEG(In)    0475 - 051C    records: 00011
```

There are no option switches.



Executing this command results in the creation of a file with the ".H00" extension and a file with the ".HEX" extension.

Extension	Description
H00	This file can be loaded into the Visual Memory Simulator. The loading time is reduced because only the code itself is saved.
HEX	The 64K-byte image of bank 0 in flash memory is stored in this file. No matter how small the program is, a 64K file is created. Whatever portion that is not filled by the program is filled with "00H". This file can be loaded into the Visual Memory Simulator, but it will not function properly.

---

**Caution:** Normally, only the H00 file is used.

---

Once the H00 file has been created, an operation check is performed in the Visual Memory Simulator. However, because the Visual Memory Simulator does not have the same clock as the actual machine, a timing check is not appropriate.

Divide the debugging phase so that the program logic is checked in the simulator and the timing and speed are checked on the actual machine.

---

**Reference:** For details on the Visual Memory Simulator, refer to the "Visual Memory Simulator Guide."

---

## Converting a HEX File to a Binary File

This procedure uses H2BIN.EXE to convert an H00 file that was created by the E2H86K into a binary file (extension ".BIN").

Input the following command line.

```
H2BIN TEST.H00 TEST.BIN
```

There are no option switches. The second parameter "TEST . BIN" may be omitted. If it is omitted, the extension ".BIN" is automatically used.

This procedure creates a file that can be loaded into Visual Memory on the actual machine.

## Creating a MAKE File

If a MAKE file is created for the MAKE command, it is possible to perform the assembly, linking, and file format conversion processes through batch processing.

If the dependence information, such as which files to insert in which commands and which files are output, is described in the MAKE file and the MAKE command is executed, the command compares the time stamps of the files that are to be inserted and are to be output, and then assembles and links only those files that have been updated.

For details on the MAKE command, refer to the "Visual Memory Programmer's Manual."

---

**Caution:** Because the MAKE command is provided for a variety of development environments, when the computer that you are using has multiple development environments installed, either change the command retrieval path (the environment variable "PATH") or change the file name of the MAKE command.

---

## Visual Memory Unit (VMU) Tutorial Revision

---

The following file is an example of the type of MAKE file to create in order to MAKE the series of procedures described up to this point. For this example, we will assume that file name is "TEST.MAK".

```
TARGET = test
OBJECTS = ifork.obj test.obj
HEADOBJ = ghead.obj
SYSOBJ = $(TOOL86)\obj\gdummy.obj

.asm.obj:
m86k $*

$(TARGET).eva: $(HEADOBJ) $(OBJECTS)
186k $(HEADOBJ) $(SYSOBJ) $(OBJECTS),$(TARGET).eva,,,

$(TARGET).h00: $(TARGET).eva
e2h86k $(TARGET)

$(TARGET).hex: $(TARGET).h00
h2bin $(TARGET).h00
```

This MAKE file is built by specifying MAKE as shown below. This assembles and builds the source files that have been updated.

---

**Caution:** Specify the "/F" option when executing the MAKE file. Input the command line in this format: "MAKE /F <MAKE file name>".

---

```
D>MAKE /F TEST.MAK

SANYO LC86000 Series MAKE Utility Version 1.00A
Copyright (C) SANYO Electric Co.,Ltd. 1993-1994 All rights reserved.

m86k GHEAD

SANYO (R) LC86K series Macro Assembler Version 4.0K
Copyright (c) SANYO Electric Co., Ltd. 1989-1995. All rights reserved.

Pass 1 .....
Source file:          GHEAD
Chip name:           LC868700
ROM size:            60K bytes
RAM size:            512 bytes
XRAM size:           196 bytes
Pass 2 .....

m86k IFORK

SANYO (R) LC86K series Macro Assembler Version 4.0K
Copyright (c) SANYO Electric Co., Ltd. 1989-1995. All rights reserved.
```

(Subsequently, the Linker, E2H86K, and then H2BIN are executed and a binary file is created.)

## Creating the Information Fork

In the sample in the Visual Memory SDK, IFORK.ASM is created and then a binary file is created. It is also possible to use a binary editor, etc., to fill the information fork of the binary file that was produced.

The information fork is filled with the icons and application names that are displayed on the Dreamcast file management screen, the comments that are displayed in Visual Memory file mode, the game names (sort keys), and comments that use larger icons.

Of these, the required items are VM comment data, GUI comment data, game names, the number of icons, visual types, and icon information (for a minimum of one icon).

Refer to chapter on, “Interfacing between Visual Memory and Dreamcast,” while editing the information fork.

## Transferring the Program to Visual Memory

Once editing of the information fork is complete, transfer the file to Visual Memory. The “Memory Card Utility” that is provided in the Visual Memory SDK is used to transfer the program.

---

**Caution:** The Memory Card Utility is provided in ELF file format as a Dev.Box application. It is not a program for general-purpose personal computers.

---

The following hardware and software is needed in order to transfer a program to Visual Memory:

- 1) An RS-232C cross cable
- 2) A communications program that runs under Windows
- 3) A debugger, such as CodeScape
- 4) GD Workshop
- 5) Dev.Box (Set 5.2X or later)

Note that items 1) and 2) are not provided in our SDK, and must be obtained separately.

---

**Reference:** For details on the transfer method, refer to chapter on, “Memory Card Utility.”

---



# ***Interfacing between Visual Memory and Dreamcast***

---

This chapter describes the interface between the Dreamcast startup screen with Visual Memory. This interface is installed in boot ROM, and is deeply inter-related with the menu screen that is displayed when Dreamcast is started up.

Following the explanation of the menu screen is a description of the data structure of the file that implements that menu screen.



---

**Caution:** The descriptions in this manual are based on boot ROM version 1.001. Some functions or names may be different in future upgrades.

---

Note that the version number is not displayed in the upper right corner on the actual machine.

# **Names of Elements in the Startup Screen**

After the opening animation that is displayed when the Dreamcast power is turned on, the screen shown on the previous page is displayed. This screen is called the "Main Menu." The Main Menu is controlled and administered by the boot ROM in Dreamcast.

## **Memory Selection Screen**

If a file for which Visual Memory is displayed is selected from the Main Menu, the following screen appears.

This screen is called the "Memory Selection Screen." As of November 30, 1998, the term "memory" refers to "Visual Memory," but other storage media may be available in the future.

When there is more than one controller or memory module connected to Dreamcast or a controller, a list of these devices is displayed.

---

**Caution:** Devices other than storage media, such as a voice recognition device, are not displayed.

---

When you look at the list, you will notice some icons that have an image of a monster in them, some that have an image of an animal, and others that are empty. Unique icons can be assigned to each memory module. (One icon per module.)

There are two types of icons: those that the end user uniquely assigns, and those that are displayed by writing a special file in Visual Memory. The icon in the lower center indicates memory that has not been initialized.

### **Label Icons**

Those icons that the end user uniquely assigns are called "label icons." when memory has been initialized, the user can freely assign any of the 124 icons stored in boot ROM. In the case of Visual Memory, the same icon that is displayed on the screen is also displayed on the LCD on the Visual Memory unit.

Note that when both a label icon and a volume icon (explained later) are assigned to one memory module, the volume icon takes precedence and is displayed.

---

**Reference:** The pattern data for the icons in boot ROM can be read by using the boot ROM font function. For details on the boot ROM font function, refer to the "Sega Library Manual Vol. 2." For a list of labels, refer to the Appendix, "List of Label Icons."

---

### **Volume Icons**

The monster icons are called "volume icons." Volume icons can be implemented by storing a file called "ICONDATA\_VMS" in Visual Memory. Accordingly, the end user cannot assign volume icons.

These icons are 32 x 32 graphic images that use 16 colors out of a possible 65,536 (ARGB4444). In the case of Visual Memory, when the unit is connected to a Dreamcast controller, the same graphic as the volume icon that is displayed on the screen is displayed on the LCD of the Visual Memory unit in monochrome. In order to display volume icons, the file "ICONDATA\_VMU" must be prepared and the Memory Card Utility must be used to transfer that file into memory.

---

**Reference:** For details on how to transfer the volume icon file, refer to the Chapter on, "Memory Card Utility."

---

### File Management Screen

Once a memory module is selected from the Memory Selection screen, the following screen is displayed. This screen is called the “File Management Screen.”



This screen displays a list of the applications that are stored in the memory module that was selected, and a list of the Dreamcast game save data.

When a file is selected, detailed information on that file is displayed on the bottom portion of the screen.

#### ① Body Color

A single color can be assigned to a single memory module. This color can be specified when initializing the memory. The end user can also change the color. Note that because the color information is stored within Visual Memory, it cannot be changed through Dreamcast in real time.

#### ② Animated Icon

The 32 x 32 icons can be displayed with a graphic that uses 16 colors out of a possible 65,536. Data for up to three patterns can be used to display animation.

One icon represents one file. When an icon is selected, the border flashes yellow and detailed information on the selected file is displayed on the bottom portion of the screen. Application files are displayed with green borders and data files are displayed with black borders.

---

**Note:** When multiple files are selected by using the X button and the Y button, the GUI comments and the total number of blocks for all of the selected files are displayed.

---

#### ③ GUI Comments

GUI comments can be displayed using normal-width letters numbers, symbols, and kana, and double-width characters. The character string length is 32 bytes. The normal-width characters that can be displayed are ASCII codes 20H to 7EH and 0A1H to 0DFH. The double-width characters are the Shift JIS codes.

---

**Note:** IS Level 2 characters are also supported. JIS X 0208-1983 is supported, so musical notes and other symbols can also be displayed.

---

## Visual Memory Unit (VMU) Tutorial Revision

### ④ File Names

A list of the files that are stored in that memory module is displayed. The characters that are not shaded in the following table can be used in file names.

		Lower 4 bits															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Upper 4 bits	0			Space	0	@	P	'	p					ー	タ	ミ	
	1			!	1	A	Q	a	q				。	ア	チ	ム	
	2			"	2	B	R	b	r				「	イ	ツ	メ	
	3			#	3	C	S	c	s				」	ウ	テ	モ	
	4			\$	4	D	T	d	t				、	エ	ト	ヤ	
	5			%	5	E	U	e	u				・	オ	ナ	ユ	
	6			&	6	F	V	f	v				ヲ	カ	ニ	ヨ	
	7			'	7	G	W	g	w				ア	キ	ヌ	ラ	
	8			(	8	H	X	h	x				イ	ク	ネ	リ	
	9			)	9	I	Y	i	y				ウ	ケ	ノ	ル	
	A			*	:	J	Z	j	z				エ	コ	ハ	レ	
	B			+	:	K	[	k	[				オ	サ	ヒ	ロ	
	C			,	<	L	¥	l					ヤ	シ	フ	ワ	
	D			-	=	M	]	m	}				ユ	ス	ヘ	ン	
	E			.	>	N	^	n	~				ヨ	セ	ホ	^	
	F			/	?	O	_	o					ッ	ソ	マ	°	

Characters That Can Be Used in File Names

**Note:** Lower-case letters may not be used. Although a "-" can be input in the Memory Card Utility, do not use this character in file names. Such an application will not be in conformance with the software creation standards.

### ⑤ VM Comments

These are comments that are displayed in Visual Memory file mode. In addition to the characters that can be used in file names, lower-case letters can also be used in VM commands.

### ⑥ Save Time

The date and time at which a file was saved are displayed. This data cannot be changed from within an application.

### ⑦ Number of Blocks Used

The file size is shown in blocks. Since one block is 512 bytes, in the case of Visual Memory (HKT-7000) a maximum of 200 blocks can be used for data storage, and a maximum of 128 blocks can be used for the game. This data cannot be changed from within an application.



## ⑧ Data Type

This indicates whether the file in question is an application or Dreamcast save data. This data cannot be changed from within an application.

---

**Caution:** Changes are possible only when the Memory Card Utility was used.

---

## ⑨ Visual Comments

A 72 x 56 graphic that uses up to 65,536 colors (ARGB4444) can be displayed. It is also possible to not display visual comments.

## Creating a Volume Icon

In order to display a volume icon on the Memory Selection Screen, create a file named `ICONDATA.VMU` with the file specifications described below.

---

**Reference:** In the Visual Memory SDK, samples are contained in the "Volumeicon" folder. The assembly source code is also provided for reference.

---

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
0000	VM comment data															
0010	Monochrome icon data start address				Color icon data start address				Reserved							
0020	Monochrome icon pattern data (32 x 32 bits)															
0030																
0040																
0050																
0060																
0070																
0080																
0090																
00A0	Color icon palette data (16 colors)															
00B0																
00C0	Color icon pattern data (32 x 32 bits)															
00D0																
00E0																
00F0																
0100																
0110																
0120																
0130																
0140																
0150																
0160																
0170																
0180																
0190																
01A0																
01B0																
01C0																
01D0																
01E0																
01F0																
0200																
0210																
0220																
0230																
0240																
0250																
0260																
0270																
0280																
0290																
02A0																
02B0																

## VM Comment Data

This is filled with a 16-byte comment. This is displayed when `ICONDATA.VMU` is selected on the Visual Memory File Management Screen or the Dreamcast File Management Screen.

Comments are displayed in Visual Memory file mode. See "File Management Screen" on page 11 for the characters that can be used in a VMU comment. Fill any unused bytes with the space character (20H).

## Visual Memory Unit (VMU) Tutorial Revision

---

### Monochrome Icon Data Start Address

This specifies the starting address of pattern data for a monochrome icon as an offset address from the start of the file.

Normally, this data is 00000020H. ("20 00 00 00" in a memory dump.)

---

**Caution:** Specify the data in Little Endian format. For details, see appendix.

---

### Color Icon Data Start Address

This specifies the starting address of palette data for a color icon as an offset address from the start of the file.

Normally, this data is 000000A0H. ("A0 00 00 00" in a memory dump.)

---

**Caution:** Specify the data in Little Endian format. For details on Little Endian format, refer to the Appendix, "Little Endian Format."  
Note that this offset address points to the starting address of the palette data for color icons, not the pattern data.

---

### Reserved Area

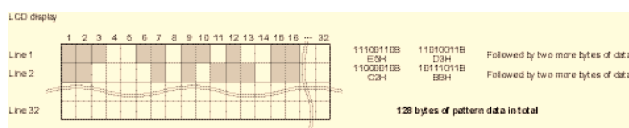
This area is reserved for future expansion. Fill this eight-byte area with "00".

### Monochrome Icon Pattern Data

This data specifies the 32 x 32-dot monochrome volume icon that is displayed on the LCD of the Visual Memory unit while the Memory Selection Screen or the File Management Screen is displayed.

This data is pattern data, starting from the upper right of the LCD and heading towards the lower left. One byte contains the pattern data for eight dots. The MSB of the data is the left-hand bit, and the LSB is the right-hand bit. Setting a bit to "1" causes the corresponding dot to be displayed as black (blue) on the LCD.

One line (32 dots) requires four bytes of data, so the 32-dot x 32-line pattern requires 128 bytes of pattern data.



### Color Icon Palette Data

Color icons can be displayed with a 16-color graphic. Write the 16-color palette in this area with ARGB4444 palette data. Each palette data entry consists of two bytes.

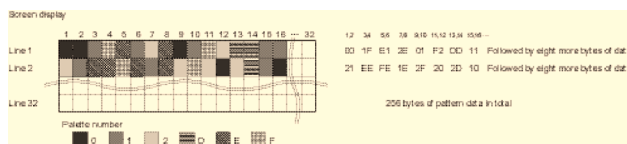
A value from "0" to "0FH" can be specified for each of A, R, G, and B. Note that a value of "0" for A makes the color transparent, and a value of "0FH" makes the color opaque.

## Color Icon Pattern Data

This is the pattern data for the icon that is displayed on the Memory Selection Screen or the File Management Screen.

This data is specified with palette numbers, starting from the upper right and heading towards the lower left. Four bits of data specify the palette number for one dot. The upper four bits are the palette number for the right-hand dot, and the lower four bits are the palette number for the left-hand dot.

One line consists of 32 dots, which requires 16 bytes of data, so the 32-dot x 32-line pattern requires 512 bytes of pattern data.



## Creating an Animated Icon

The data for animated icons and GUI comments that are displayed on the File Management Screen must be included in the files themselves. Because there are three different file structures, this section explains the file types and the file structures.

### Three File Structures

There are three files stored in memory, and each has its own file structure.

#### ICONDATA\_VMS Format

This is the structure that was described in the previous section. This file does not have an information fork.

---

**Note:** For details, refer to "Section , "Creating a Volume Icon"

---

#### Visual Memory Application Format

Files that can be executed in Visual Memory game mode must have the following file structure:

Address	Contents
0000	Visual Memory header (equivalent to GHEAD.ASM)
0200	Information fork
xxxx	Application code

## Visual Memory Unit (VMU) Tutorial Revision

---

### Data File Format

A file that stores game data for a Dreamcast application must have the following file structure:

Address	Contents
0000	Information fork
0200	Game data

### Information Fork

Files other than the "ICONDATA\_VMU" file have a section called an "information fork." This information is the colored portion of a file dump displayed by the Memory Card Utility. All detailed file information is contained in this information fork.

---

**Caution:** The data area for an animated icon is not displayed in color.

---

**Note:** Refer to the Visual Memory SDK sample "total", since it includes an information fork for an animated icon and visual comments.

---

The structure of the information fork is described below. Note that the addresses in the table are given as offset addresses from the start of the information fork.

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
0000	VM comment data															
0010	GUI comment data															
0020																
0030	Game name (soft key)															
0040	Number of icons	Animation speed	Visual type	CRC	Save data size	Reserved										
0050	Reserved															
0060	Icon palette data (16 colors)															
0070																
0080	Icon #1 pattern data (32 x 32 data)															
0090																
0100																
0260																
0270																
0280	Icon #2 pattern data (32 x 32 data)															
0290																
0300																
0460																
0470																
0480	Icon #3 pattern data (32 x 32 data)															
0490																
0500																
0660																
0670																
0680	Visual comment (palette and pattern data)															
0690																
XXXX																

### VM Comment Data

This contains a 16-byte comment. This comment is displayed when the file is selected on the Visual Memory File Management Screen or the Dreamcast File Management Screen.

The comment is displayed in Visual Memory File Mode. The characters that are not shaded in the following table can be used in VM comments. Note that any bytes that are not used should be filled with space characters (20H).

		Lower 4 bits																			
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F				
Upper 4 bits	0			Space	0	@	P	^	p									ー	タ	ミ	
	1			!	1	A	Q	a	q									。	ア	チ	ム
	2			"	2	B	R	b	r									「	イ	ツ	メ
	3			#	3	C	S	c	s									」	ウ	テ	モ
	4			\$	4	D	T	d	t									、	エ	ト	ヤ
	5			%	5	E	U	e	u									・	オ	ナ	ユ
	6			&	6	F	V	f	v									ヲ	カ	ニ	ヨ
	7			'	7	G	W	g	w									ア	キ	ヌ	ラ
	8			(	8	H	X	h	x									イ	ク	ネ	リ
	9			)	9	I	Y	i	y									ウ	ケ	ノ	ル
	A			*	:	J	Z	j	z									エ	コ	ハ	レ
	B			+	:	K	[	k										オ	サ	ヒ	ロ
	C			,	<	L	¥	l										ヤ	シ	フ	ワ
	D			-	=	M	]	m	}									ユ	ス	ヘ	ン
	E			.	>	N	^	n	~									ヨ	セ	ホ	°
	F			/	?	O	_	o										ッ	ソ	マ	°

*Characters That Can Be Used in VM Comments*

### GUI Comment Data

This contains a 32-byte comment that is displayed on the File Management Screen. The comment is displayed when the file is selected on the Dreamcast File Management Screen.

The normal-width characters that can be used are those that are not shaded in the following table. The double-width characters are the Shift JIS codes; JIS Level 2 characters are also supported. JIS X 0208-1983 is supported, so musical notes and other symbols can also be displayed.

Note that any bytes that are not used should be filled with space characters (20H).

		Lower 4 bits																			
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F				
Upper 4 bits	0			Space	0	@	P	^	p									ー	タ	ミ	
	1			!	1	A	Q	a	q									。	ア	チ	ム
	2			"	2	B	R	b	r									「	イ	ツ	メ
	3			#	3	C	S	c	s									」	ウ	テ	モ
	4			\$	4	D	T	d	t									、	エ	ト	ヤ
	5			%	5	E	U	e	u									・	オ	ナ	ユ
	6			&	6	F	V	f	v									ヲ	カ	ニ	ヨ
	7			'	7	G	W	g	w									ア	キ	ヌ	ラ
	8			(	8	H	X	h	x									イ	ク	ネ	リ
	9			)	9	I	Y	i	y									ウ	ケ	ノ	ル
	A			*	:	J	Z	j	z									エ	コ	ハ	レ
	B			+	:	K	[	k										オ	サ	ヒ	ロ
	C			,	<	L	¥	l										ヤ	シ	フ	ワ
	D			-	=	M	]	m	}									ユ	ス	ヘ	ン
	E			.	>	N	^	n	~									ヨ	セ	ホ	°
	F			/	?	O	_	o										ッ	ソ	マ	°

*Normal-width Characters That Can Be Used in GUI Comments*

## Visual Memory Unit (VMU) Tutorial Revision

---

### Game Name (Sort Key)

File names are sorted when they are listed on the Dreamcast File Management Screen. This area is used for the sort key. 16 bytes of data are specified for this area; fill this area with unique character code display data.

---

**Reference:** For details on assigning game names and the unique code table, refer to Section , "Game Name Sorting Rules".

---

### Number of Icons

For an animated icon, specify either "2" or "3" in this field. Specify "1" for a still-image (normal) icon.

The range of values that can be specified in this field is "1" to "3," so an animation pattern can consist of a maximum of three patterns.

---

**Caution:** Do not specify a value outside the range of "1" to "3". Operation is not guaranteed if "0" or a value of "4" or more is specified.

---

Specify the data in Little Endian format. For details on Little Endian format, refer to the Appendix, "Little Endian Format."

### Animation Speed

This specifies the speed at which the icons are switched when using an animated icon. The range of values that can be specified is from "1" to "65,535." If the specified value is "n," the animation patterns are switched every  $n/30$  seconds.

For example, if "1" is specified, the animation patterns are switched every  $1/30$  of a second. If "30" is specified, the animation patterns are switched every second. If "65,535" is specified, the animation patterns are switched roughly every 36 minutes.

When there are three animation patterns, they are displayed in the sequence  $1 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow \dots$ . If there are two animation patterns, they are displayed alternately. If there is only one icon, this value is meaningless.

---

**Caution:** Do not specify a value of "0." Operation is not guaranteed if "0" is specified.

---

Specify the data in Little Endian format. For details on Little Endian format, refer to the Appendix, "Little Endian Format."

### Visual Type

This specifies the type of the visual comment that is displayed in the lower right corner of the Dreamcast File Management Screen. A 72 x 56-dot graphic can be displayed for a visual comment.

---

**Caution:** Animation cannot be used for the visual comment. Also note that using a graphic with a lot of colors will consume more memory.

---

Specify the data in Little Endian format. For details on Little Endian format, refer to the Appendix, "Little Endian Format."

The specifiable values and the corresponding visual comment types, number of bytes required, and number of blocks used are listed in the following table.

Specified value	Visual comment type	Number of bytes required	For data	For palette	Number of blocks used
0	None	0	0	0	0
1	Direct color (type A)	8064	8064	0	16
2	256-color graphic (type B)	4544	4032	512	9
3	16-color graphic (type C)	2048	2016	32	4

---

**Reference:** For details on visual comments, refer to section 2.3.3, "Visual Comment Data Structure."

---

### **CRC**

Write the CRC (error checking/correction code) in this field.

When a file is saved by using the backup utility function `"buMakeBackupFileImage()"` from the Sega Library, the CRC is calculated automatically, and that value is written in this field. Note that the CRC applies only to the data portion, and not to the information fork.

---

**Caution:** When a Visual Memory application is transferred by using the Memory Card Utility, there is no need to write the CRC value or to perform a CRC check. In this case, fill this field with "00 00." Specify the data in Little Endian format. For details on Little Endian format, refer to the Appendix, "Little Endian Format."

---

### **Save Data Size**

Specify the size of the data area (not including the information fork) in bytes.

---

**Caution:** When a Visual Memory application is transferred by using the Memory Card Utility, there is no need to write the data size. In this case, fill this field with "00 00."

---

Specify the data in Little Endian format. For details on Little Endian format, refer to the Appendix, "Little Endian Format."

### **Reserved Area**

This area is reserved for future expansion. Fill this 20-byte area with "00."

### **Icon Palette Data**

This specifies the 16 colors of the palette that is used for the icon pattern that follows.

Specify the palette data in ARGB4444 format. Specify two bytes for each color.

A value from "0" to "0FH" can be specified for each of A, R, G, and B. Note that a value of "0" for A makes the color transparent, and a value of "0FH" makes the color opaque.

This palette also is used for icons #2 and #3. Note that it is not possible to change the palette for each icon.

# Visual Memory Unit (VMU) Tutorial Revision

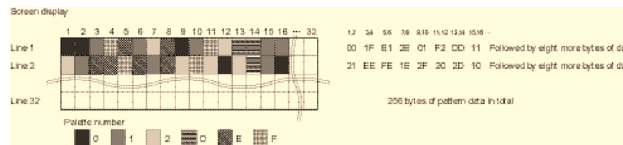
## Icon #n Pattern Data

This is the pattern data for a 32 x 32-dot, 16-color icon.

This data is specified with palette numbers, starting from the upper right and heading towards the lower left. Four bits of data specify the palette number for one dot. The upper four bits are the palette number for the right-hand dot, and the lower four bits are the palette number for the left-hand dot.

One line consists of 32 dots, which requires 16 bytes of data, so the 32-dot x 32-line pattern requires 512 bytes of pattern data.

If animation is not to be used, create pattern data for one pattern. If animation is to be used, provide pattern data for up to three patterns.



## Visual Comment Data

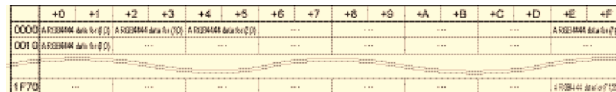
When using a visual comment (when a value other than "0" was specified for the visual type), specify the data in this field. The visual comment data is the data for a 72 x 56-dot graphic.

## Visual Comment Data Structure

The visual types and the visual comment data structures are described below.

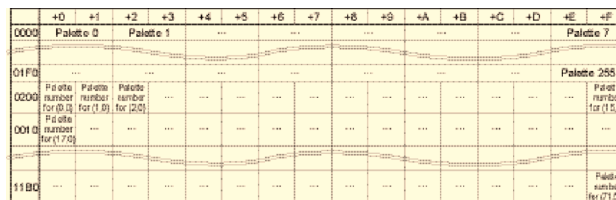
### Direct Color (Type A)

This type has no palette data. Specify the palette data using ARGB4444 values. Two bytes of data are used for one dot.



### 256-color Graphic (Type B)

This has palette data for 256 colors. The palette colors are specified in ARGB4444. The pattern data is specified by specifying a palette number (one byte) for each dot.



### 16-color Graphic (Type C)

This has palette data for 16 colors. The palette colors are specified in ARGB4444.

The pattern data is specified with four bits for each dot. One byte contains data for two dots; the upper four bits specify the palette number for the left-hand dot, and the lower four bits specify the palette number for the right-hand dot.



### **Game Name Sorting Rules**

The game name (sort key) in the information fork is used to determine the order in which icons are displayed on the Dreamcast File Management Screen. Therefore, it is necessary to create a game naming scheme that will cause titles of games that are a series of sequels to be displayed in order.

*Example:*

"DreamPassport01" DreamPassport02"



# ***Memory Card Utility***

---

This chapter explains how to use the Memory Card Utility program, which is used to transfer files between a PC and the Dev.Box, and between the Dev.Box and Visual Memory.

## **Memory Card Utility Preparation and Startup**

The Memory Card Utility is a Dreamcast program. A file called "mem\_util.elf" is provided in the "utility" folder in the folder where the Visual Memory SDK was installed. Executing this program requires a program that is included in the Dreamcast SDK, such as CodeScape or GD Workshop.

### **Requirements for Transfer**

The Memory Card Utility exchanges files with a PC through an RS-232C serial interface. Therefore, the following items are required in addition to the SDK provided by Sega:

- 1) An RS-232C reverse cable
- 2) A communications program that runs under Windows

Item 1) can be purchased at most computer stores. The cable should be connected between the RS-232C interface on the PC and the connector labeled "SERIAL" on the back of the Dev.Box.

*RS-232C Serial Interface Connector on the Computer*

*SERIAL Connector on the Dev.Box*

---

**Caution:** Be certain to use a reverse cable. A straight cable will not work.

---

"HyperTerminal," which is provided with Windows, suffices for item 2). If you use other communications software, it must support file transfers using the Xmodem protocol.

If you do not have HyperTerminal in your Windows environment, you can install it by following this procedure:

- 1) In the "Control Panel" window, double-click on the "Add/Remove Programs" icon.
- 2) Click the "Windows Setup" tab.
- 3) Double click "Communications".
- 4) Check the box next to "HyperTerminal."

HyperTerminal will now be installed in your system.

---

**Note:** The version of HyperTerminal that is provided with Windows 98 may not be able to recognize path and file names correctly if they contain Kanji characters, or a file with corrupted characters may be stored in the next higher folder. In this case, we recommend either not using file names that contain Kanji characters, or else using another communications program.

---

Also prepare the following items, which are included in Sega's SDK:

- 1) Dev.Box (Set 5.2X or higher)
- 2) CodeScope
- 3) GD Workshop
- 4) Visual Memory
- 5) Dreamcast controller

For details on connections and setup, refer to the "Setup Guide." The explanations in this manual will assume that setup has been completed properly.

---

**Caution:** The Memory Card Utility will not run on a Set 5.1X or earlier Dev.Box.

---

## Software Preparation

Once the PC has been connected to the Dev.Box through the RS-232C interface and software setup has been completed, it is necessary to set the properties, etc.

### Communications Protocol Setup

This setting specifies which communications profile is to be used for transferring data between the PC and the Dev.Box.

Make this setting in the application that you will be using. Refer to the manual for that application for details on making that setting.

---

**Caution:** When the Memory Card Utility is started up, a startup message is displayed on the communications screen. If this message is not displayed correctly, recheck the communications protocol and make sure that the interface to which the cross cable is connected is selected.

---

Setting Item	Setting
Communications speed	38,400bps
Data length	8 bits
Stop bit length	1 bit
Parity checking	None
Flow control	None
Kanji codes	Shift JIS codes

When using HyperTerminal, set the dialog boxes as shown below.

---

**Caution:** Flow control must be set to "None." Do not specify "Xon/off" or hardware control.

---

If there are multiple RS-232C interfaces, select the interface to which the cross cable is connected.

### **GD Workshop Setup**

At startup, the Memory Card Utility detects the door on the GD-ROM drive being closed (i.e., that a GD-ROM is mounted). Therefore, it is necessary to create a dummy GD-ROM to emulate the door being closed.

---

**Reference:** For details on the operation of GD Workshop, refer to the "GD Workshop Manual."

---

Create a dummy GD-ROM according to the procedure described below.

- 1) Start up GD Workshop.
- 2) Create a new project. For this example, we will create a project named "DoorClose."
- 3) Drag files suitable for three tracks.  
Since emulation is not possible if the file is too small, copy an execution file for a large application, etc.  
For example, drag the warning sound file that is supplied with the Dreamcast SDK to an audio track.
- 4) Save the project.

Confirm that the dummy GD-ROM was created properly.

- 5) Start up DA Checker and restart the Dev.Box in OS mode.
- 6) Start up GD Workshop, and load the project named "DoorClose."
- 7) Select "Sound" from the Dreamcast main menu.
- 8) In GD Workshop, change the GD-ROM to emulation mode.
- 9) Press the "Door Close/Open" button to close the GD-ROM door and mount the GD-ROM.  
If the dummy GD-ROM has been created properly, the CD-ROM graphic will appear in the Sound menu. If the "CD-ROM" is played back, the warning sound should be heard.

In the future, dummy GD-ROM emulation can be initiated simply by loading the project.

### **Memory Card Utility Startup**

This section describes how to execute the Memory Card Utility using CodeScape.

- 1) Before starting up the Memory Card Utility, start up GD Workshop.  
Once GD Workshop has been started up, load the "DoorClose" project and start initialization. Press the "Door Close/Open" button to mount the GD-ROM.
- 2) Start up the communications software. Confirm the settings for the communications protocol, the interface number, etc.
- 3) Start up CodeScape.
- 4) Select "Load Program File..." from the "File" menu.
- 5) Click the [Browse] button and select "mem.util.elf".  
The file "mem.util.elf" in the "utility" folder in the folder where the Visual Memory SDK was installed.
- 6) Select the "Load Binary Only" option.
- 7) Check the "Enable Reset Options" box and select the "Perform Hard Reset of Target" option.
- 8) Check the "Enable Run Options" box and select the "Run" option.
- 9) Once the dialog box settings have been made, click the [OK] button.
- 10) A graph bar is displayed; when it reaches 100%, the Memory Card Utility starts up. At this point, the following startup message is displayed in the communications software window:

If the message does not appear or if it is corrupted, recheck the communications protocol settings.

The main menu of the Memory Card Utility appears on the Dev.Box display (an NTSC or VGA monitor).

---

**Note:** If "Session Save" is selected from the "File" menu at this point, the Memory Card utility can be started up next time simply by opening the session.

---

- 11) Except for file transfers between a PC and the Dev.Box, subsequent Memory Card Utilities are performed by the Dreamcast controller.

## Memory Card Utility Operation

This section explains how to operate the Memory Card Utility. Note that the manual is based on version 0.64.1.

Operations are performed by using the direction button to select a menu item and then pressing the A button to enter that selection. Pressing the B button cancels the operation and returns you to the previous menu. On some menus, the functions of the buttons sometimes differ, or other buttons are used. In those cases, the functions of the buttons will be explained for each menu.

### Main Menu

The following screen appears when the Memory Card Utility is started up. This is called the "Main Menu."

Menu items that are grayed out are items that have not been implemented and cannot be selected.

#### **BACKUP UTILITY**

If this menu item is selected, the Memory Selection Menu is displayed.

This menu is normally selected when transferring a file to Visual Memory, or to work with a file that has already been saved.

#### **SYSTEM CONFIG**

This menu item is used to set Visual Memory's internal clock according to the Dev.Box's internal clock.

If this menu item is selected, the System Configuration Menu appears. If "MEMORY CARD TIME" is selected, a confirmation message appears and the clocks in all of the Visual Memory units that are connected are set according to the Dev.Box clock.

---

**Note:** Only "MEMORY CARD TIME" can be selected in the System Configuration Menu.

---

#### **EXIT**

This menu item terminates the Memory Card Utility and passes control to boot ROM. The Dreamcast startup menu is displayed.

### Memory Selection Menu

If "BACKUP UTILITY" is selected from the Main Menu, the Memory Selection Menu is displayed.

---

**Caution:** Visual Memory can be inserted or removed until this screen.  
When inserting or removing Visual Memory, set the operation mode to a mode other than game mode.  
If Visual Memory is connected while it is in game mode, the connection status will not be recognized correctly.

---

This menu is used to select the memory unit that is to be the object of subsequent operations.

All of the memory units that are currently connected are displayed on this screen. "A," "B," "C," and "D" represent the controller ports. Upper and lower tiers are displayed when multiple memory units are connected to one controller.

## **Visual Memory Unit (VMU) Tutorial Revision**

---

Note that in the case of a Visual Memory unit, a number that corresponds to the screen is displayed on the LCD. After selecting the desired memory unit, press the A button. The Command Selection Screen now appears.

### **USED**

Displays only the number of blocks already in use in the data storage area.

### **FREE**

Displays the number of free blocks in the data storage area.

### **GAME**

Indicates the usage status of the Visual Memory application area, in the format:

«number of blocks in use»/«maximum number of blocks that can be used by an application»

For example, a display of "128/128" indicates that an application has already been written in the Visual Memory unit. A display of "0/128" indicates that no application has been written in the Visual Memory unit.

## **Command Selection Menu**

This menu is used to transfer data to Visual Memory and to manipulate files that have been written in memory.

The right side of the screen shows a list of files written in Visual Memory. If the entire list cannot be displayed on one screen, make the file list active (move the s mark to the file list) and then press the right-hand direction button to display the rest of the list.

### **INFO**

If this menu item is selected, the following screen appears:

This screen displays information concerning the capacity of Visual Memory, body color information, volume icon information, and information on the date and time of initialization.

### **RECV DATA**

This menu item is used to transfer applications, volume icon files, and data files from a PC to a Dev.Box. If "RECV DATA" is selected, the following screen appears:

On this screen, specify the type of file that is to be received from the PC. when receiving a Visual Memory application, select "GAME BUFFER;" when receiving volume icon data, select "ICON BUFFER." When receiving a data file, select "NORMAL BUFFER."

---

**Reference:** For details on data transfers from a PC to Visual Memory, refer to section, "File Transfers from a PC to Visual Memory."

---



### SAVE DATA

This menu writes data that was previously received in the Dev.Box buffer into Visual Memory. If SAVE DATA is selected, the following screen appears.

On this screen, specify which file (buffer data) to write in Visual Memory.

---

**Caution:** "GAME BUFFER" cannot be selected for Visual Memory in which an application has already been written. Delete the application file first.

---

To write a Visual Memory application, select "GAME BUFFER;" to write volume icon data, select "ICON BUFFER." To write a data file, select "NORMAL BUFFER."

### DUPLICATE

This menu item makes an exact copy of the contents of the currently selected Visual Memory unit in another Visual Memory unit. This function is equivalent to the disk copy functions in Windows and MS-DOS.

If this menu item is selected, the following screen appears:

Once the destination Visual Memory unit has been selected, a confirmation message is displayed.

---

**Caution:** The DUPLICATE function copies the contents of flash memory exactly. All data previously saved in the destination Visual Memory unit will be lost.

---

When you select "OK," the contents of memory are copied exactly.

### DEFRAG

Selecting this menu item eliminates fragments (defrags) that develop when files are repeatedly saved and deleted.

---

**Caution:** This menu item cannot be selected for a Visual Memory unit in which an application has been written.

---

Because the FAT system is used for file management in Visual Memory, the data storage area becomes fragmented as files of different sizes are written and deleted over the course of time. This function reorganizes the data so that the fragments are eliminated.

---

**Note:** Because applications have to be allocated in a continuous area in memory, execute the DEFRAG function before transferring an application into Visual Memory.

---

If an application cannot be stored even though there is sufficient space, it is likely that fragmentation is the culprit. Execute the DEFRAG function to create continuous free space.

If "DEFRAG" is selected, a confirmation message appears and then the defragmentation processing is performed.

## **Visual Memory Unit (VMU) Tutorial Revision**

---

### **FORMAT**

If this menu item is selected, the following screen appears and the selected Visual Memory unit can be initialized.

This screen is used to specify the body color and label icon, to set the date and time of initialization, etc. After making all of the necessary settings, a confirmation message appears; selecting "OK" causes the Visual Memory unit to be initialized. All files stored in the Visual Memory unit that is being initialized will be lost.

---

**Reference:** For details on the initialization procedure, see "Initializing Visual Memory" on page 33.

---

### **UNFORMAT**

This menu item can be selected in order to create an uninitialized visual memory unit. The system BIOS determines whether a Visual Memory unit has been initialized or not by checking a management area (an upper address in bank 1 in flash memory), such as the FAT.

Just as data cannot be written on a floppy disk that has not been initialized, data and applications cannot be stored in a Visual Memory unit that has not been initialized.

---

**Caution:** Visual Memory that is purchased commercially is shipped in an initialized state. Rarely, a lot might be shipped in an uninitialized state.

---

If "UNFORMAT" is selected, the following screen appears:

If "COMPLETE" is executed, 00H is written to every address in the flash memory to put it into the uninitialized state. If "QUICK" is selected, just the management area is cleared.

---

**Caution:** The "QUICK" function is not implemented in Memory Card Utility Version 0.64.1 and therefore cannot be selected.

---

Whether "COMPLETE" or "QUICK" is used, all data stored in Visual Memory is deleted. Even if "QUICK" is selected, there is no means for salvaging the data after initialization.

### **CHECK DISK**

If this menu item is executed, the following screen appears and a check is made of FAT conformance, missing data bits, etc. This function is equivalent to the scan disk function in Windows and the CHKDSK function in MS-DOS.

However, this function does not have an error correction capability.

This function also conducts a CRC check of the information fork of each file, and checks for missing bits in files.

---

**Note:** If the CRC value was omitted from an information fork, this check returns an error for that information fork, but this error can be ignored.

---

The results of these checks are also displayed in the communications software window as shown below.

## File Operations Menu

If the s mark is moved on the file list, a file is selected, and the A button is pressed, the File Operations Menu is displayed.

Multiple files can be selected by pressing the X button after selecting each file. If the A button is pressed while multiple files are selected, the menu operation that is performed is performed on all selected files.

If the Y button is pressed, all files that are currently selected are deselected, and all files that are currently not selected are selected.

### **COPY**

If "COPY" is selected, the following screen appears, and the selected file is copied to a different Visual Memory unit.

### **DELETE**

This deletes the selected file. A confirmation message is displayed when "DELETE" is selected.

### **RENAME**

This changes a file name. This item cannot be selected when multiple files are selected. If "RENAME" is selected, the following screen appears; the file name can now be changed.

---

**Caution:** The Memory Card Utility is designed to permit the "-" character to be used in file names, but do not use the character.

---

Although it poses no problem for the debugger, do not use the "-" character in file names in final products. Such an application will not be in conformance with the software creation standards.

Use "←" and "→" to move the cursor. Select "OK" to change the file name to the new file name that was input. "CANCEL" is equivalent to the B button. If "RESET" is selected, the file name that was being input is cleared and the original file name is restored.

On this screen, the R trigger and the L trigger can be used to move the cursor, and pressing the Start button has the same effect as selecting "OK."

### **ATTRIBUTE**

If "ATTRIBUTE" is selected, the following screen appears and the file attribute (copying prohibited/ permitted) can be changed.

If the COPY FLAG is set to "FF," copying that file is prohibited. A file for which copying is prohibited can not be copied through the Dreamcast File Management Screen. If COPY FLAG is set to any other value (00 to FE), copying that file is permitted. Because any value other than FF can be used, a program can be created that uses this field to determine what generation of copying a given file is.

---

**Caution:** Note that when a file for which the COPY FLAG is set to any value from 00 to FE is copied through the Dreamcast File Management Screen, the COPY FLAG in the newly copied file is set to "00."

---

HEADER OFFSET cannot be changed.

## Visual Memory Unit (VMU) Tutorial Revision

---

### UPLOAD

This can be used to transfer a selected file to a PC.

Before selecting "UPLOAD," execute an Xmodem download using the file transfer function of the communications software. As long as the Xmodem protocol is used, it does not matter if it is CRC or 1024. When the file name input screen appears, enter an appropriate file name. The file name that is input does not have to be identical to the file name that is used in Visual Memory.

When "UPLOAD" is selected, the message "NOW LOADING..." appears and the file transfer to the PC begins.

---

**Caution:** "UPLOAD" cannot be aborted. To abort, first complete the file transfer that is in progress.  
"UPLOAD" cannot be selected while multiple files are selected.

---

### INFO

If "INFO" is selected, the message "NOW LOADING..." is displayed, and then the following screen appears.

The contents of the information fork are displayed on this screen.

Because the game name (sort key) is also displayed, this screen can be used to check the game name after it has been input.

### DUMP

If "DUMP" is selected, the message "NOW LOADING..." is displayed, and then the file dump screen appears.

The portion that is the information fork is displayed in color. A character dump is also displayed on the right-hand side.

---

**Caution:** DUMP cannot be selected when multiple files are selected.  
Kanji cannot be displayed in the character dump.

---

The following buttons can be used on the file dump screen:

Up button	Scrolls towards the beginning of the file.
Down button	Scrolls towards the end of the file.
Left button	Scrolls rapidly towards the beginning of the file.
Right button	Scrolls rapidly towards the end of the file.
L trigger	Displays the beginning of the file.
R trigger	Displays the end of the file.
X button	Each time this button is pressed, the display delimiting unit switches between BYTE, WORD (2 bytes), and DWORD (4 bytes).
Start button	Halts the file dump.

### EDIT

This is for future expansion. This menu item cannot be selected because the editing function is not implemented in Ver. 0.64.1.

## Initializing Visual Memory

This section describes the procedure for initializing Visual Memory.

- 1) Select the Visual Memory unit that is to be initialized, and then display the Command Selection Menu.
- 2) Select "FORMAT," and the following screen appears.
- 3) In the "ICON NO." field, specify the label icon number. Specifying "00" specifies the default Visual Memory icon. Specify a value from 000 to 123. Do not set a value of 124 or higher.

---

**Reference:** For a list of the label icon designs and numbers, refer to the Appendix, "List of Label Icons."

---

- 4) Set whether the body color information (which is set next) is valid or not.  
If the body color information is valid, select "ENABLE." If the body color information is invalid, select "DISABLE." If "DISABLE" is selected, the body color is white and the color information setting becomes unavailable.
- 5) Set the color information. "COLOR A" specifies the transparency. A value of "FF" is completely opaque, and a value of "00" is completely transparent.  
COLOR R, G, and B specify the intensity of the red, green, and blue components. A value of "FF" is the maximum intensity, and a value of "00" is the minimum intensity.

---

**Caution:** Note that if a value of "00" is set for COLOR A, the other colors will be transparent.

---

- 6) "OPERATION" specifies whether to initialize just the FAT, or to initialize all of memory.  
Select "QUICK" to initialize just the FAT, and select "COMPLETE" to initialize all of memory.
- 7) Select "NEXT" and the following screen appears.  
Set the date and time of initialization. Use the left and right buttons to set the date and time, and then use the up and down buttons to change the value.
- 8) Lastly, select "SET" and a confirmation message appears. Select "OK" to begin the initialization process.

---

**Caution:** If a Visual Memory unit is initialized, all files that were written in that unit are deleted. Even if "QUICK" is selected for "OPERATION," there is no means for salvaging the data after initialization.

---

# Transferring Files from a PC to Visual Memory

This section explains the procedure for transferring files from a PC to Visual Memory. In this example, HyperTerminal, which is provided with Windows, will be used as the communications software.

- 1) Select the Visual Memory unit to which the file is to be transferred, and then display the Command Selection Menu.
- 2) Select "RECV DATA," and then select the buffer for the file that is to be transferred.  
When transferring an application, select "GAME BUFFER."  
When transferring a volume icon ("ICONDATA\_VMU"), select "ICON BUFFER."  
When transferring a data file, select "NORMAL BUFFER."
- 3) When the confirmation message is displayed, select "OK."
- 4) Select the file transfer operation in the communications software. In the "Transfer" menu, select "Send File..."
- 5) Select "Xmodem" for the protocol. Then click the [Browse] button and specify the file that is to be transferred. Finally, click the [Send] button.
- 6) While the file transfer is in progress, the following screens appear.  
In the communications software screen  
In the Dev.Box screen
- 7) When the file transfer is completed, the following screen appears on the Dev.Box side. Press the A button.
- 8) Select "SAVE DATA" and specify which buffer's contents to write in Visual Memory.  
When writing an application, select "GAME BUFFER."  
When writing a volume icon ("ICONDATA\_VMU"), select "ICON BUFFER."  
When writing a data file, select "NORMAL BUFFER."

---

**Caution:** If an application has already been written in the Visual Memory unit, "GAME BUFFER" cannot be selected. Delete the old file and then select "SAVE DATA" again.

---

- 9) If a buffer other than "ICON BUFFER" was selected, the following screen appears.

Input the file name and select "OK." The file name does not have to be in "8.3" format (an eight-character file name, a period, and a three-character extension).

If a file with the same name already exists, the program asks whether or not to overwrite the old file.

The PC-to-Visual Memory file transfer process is now complete.

If an application was transferred, return to the Memory Selection Screen, disconnect the Visual Memory unit, set the Visual Memory unit to game mode, and then execute the application.

# A. Little Endian Format

When storing multiple bytes of data in memory, some CPUs use a format that starts from the high-order byte and stores it in the high-order byte in memory, while other CPUs use a format that starts from the high-order byte and stores it in the low-order byte in memory.

For example, when storing the data "00 FE 2E EF" in memory as an "unsigned long int" value (an unsigned 32-bit integer), the following two methods could be used:

**Table A.1 Big Endian Format**

	+00	+01	+02	+03
0000	00	FE	2E	EF

**Table A.2 Little Endian Format**

	+00	+01	+02	+03
0000	EF	2E	FE	00

This format, in which the upper and lower bytes of data are stored in reverse order is called "Little Endian Format." The format in which data is stored in its normal order is called "Big Endian Format."

Because the SH4 CPU that is inside the Dreamcast uses Little Endian format, all data other than byte data must be stored in memory in Little Endian format.

For example, because the value "00 00 00 20" must be specified for the "monochrome icon data starting address" in ICONDATA\_VMU", the value is stored in memory in the order "20 00 00 00". In addition, when storing unsigned 16-bit data ("00 FF") in memory, it should be stored in the order "FF 00"





## ***B. List of Label Icons***

---

The label icons that are built into the Dreamcast boot ROM are listed on the following page.

## B. List of Label Icons

	00	0		1F	31		3E	62	<b>C</b>	5D	93
	01	1		20	32		3F	63	<b>D</b>	5E	94
	02	2		21	33		40	64	<b>E</b>	5F	95
	03	3		22	34		41	65	<b>F</b>	60	96
	04	4		23	35		42	66	<b>G</b>	61	97
	05	5		24	36		43	67	<b>H</b>	62	98
	06	6		25	37		44	68	<b>I</b>	63	99
	07	7		26	38		45	69	<b>J</b>	64	100
	08	8		27	39		46	70	<b>K</b>	65	101
	09	9		28	40		47	71	<b>L</b>	66	102
	0A	10		29	41		48	72	<b>M</b>	67	103
	0B	11		2A	42		49	73	<b>N</b>	68	104
	0C	12		2B	43		4A	74	<b>O</b>	69	105
	0D	13		2C	44		4B	75	<b>P</b>	6A	106
	0E	14		2D	45		4C	76	<b>Q</b>	6B	107
	0F	15		2E	46		4D	77	<b>R</b>	6C	108
	10	16		2F	47		4E	78	<b>S</b>	6D	109
	11	17		30	48		4F	79	<b>T</b>	6E	110
	12	18		31	49		50	80	<b>U</b>	6F	111
	13	19		32	50	<b>0</b>	51	81	<b>V</b>	70	112
	14	20		33	51	<b>1</b>	52	82	<b>W</b>	71	113
	15	21		34	52	<b>2</b>	53	83	<b>X</b>	72	114
	16	22		35	53	<b>3</b>	54	84	<b>Y</b>	73	115
	17	23		36	54	<b>4</b>	55	85	<b>Z</b>	74	116
	18	24		37	55	<b>5</b>	56	86		75	117
	19	25		38	56	<b>6</b>	57	87		76	118
	1A	26		39	57	<b>7</b>	58	88		77	119
	1B	27		3A	58	<b>8</b>	59	89		78	120
	1C	28		3B	59	<b>9</b>	5A	90		79	121
	1D	29		3C	60	<b>A</b>	5B	91		7A	122
	1E	30		3D	61	<b>B</b>	5C	92		7B	123

## ***C. Sample Program Listings***

---

This section includes listings of the sample programs that are included with the Visual Memory SDK. Explanations of the information fork source file "IFORK.ASM" and details of "GHEAD.ASM" are not included.

---

**Caution:** When viewing the sample programs, set the tab (09H) to "4" (byte).

---

## LCD Pattern Display

This sample program displays a simple pattern on the LCD (XRAM).

Lines 51, 55, 59, 63, and 67 specify the pattern, and the "matrix" routine draws this pattern on the LCD.

```
001 ; Tab width = 4
002
003 ;-----
004 ; ** LCD display processing sample 1 **
005 ;
006 ; Transfers data to display RAM and displays a simple pattern on the display
007 ;-----
008 ; 1.00 981208 SEGA Enterprises,LTD.
009 ;-----
010
011 chip Lc868700 ; Specifies the chip type for the assembler
012 worldexternal ; External memory program
013
014 publicmain ; Symbol referenced from ghead.asm
015
016 extern_game_end ; Application end
017
018
019 ; **** Definition of System Constants ****
020
021 ; OCR (Oscillation Control Register) settings
022 osc_rcequ 081h ; Specifies internal RC oscillation for the System clock
023 osc_xtequ 082h; Specifies crystal oscillation for the system Clock
024
025
026 ; *** Data Segment ****
027
028 dseg ; Data segment start
029
030 r0: ds 1 ; Indirect addressing register r0
031 r1: ds 1 ; Indirect addressing register r1
032 r2: ds 1 ; Indirect addressing register r2
033 r3: ds 1 ; Indirect addressing register r3
034 ds 12 ; Other registers reserved for the system
035
036
```

```
037 ; *** Code Segment *****
038
039     cseg           ; Code segment start
040
041 ; *-----*
042 ; * User program           *
043 ; *-----*
044 main:
045     mov     #0f0h,c    ; Display data
046     call   matrix     ; Display pattern on the LCD
047     set1   PCON,0     ; Enters HALT mode and waits for an interrupt.
048                     ; HALT mode is cancelled and processing continues
049                     ; when a base timer interrupt is generated.
050
051     mov     #00fh,c    ; The following lines display different patterns in the same manner
052     call   matrix
053     set1   PCON,0
054
055     mov     #0cch,c
056     call   matrix
057     set1   PCON,0
058
059     mov     #033h,c
060     call   matrix
061     set1   PCON,0
062
063     mov     #055h,c
064     call   matrix
065     set1   PCON,0
066
067     mov     #0aah,c
068     call   matrix
069     set1   PCON,0
070
071                                     ; ** [M] (mode) Button Check **
072     ld     P3
073     bn     acc,6,finish    ; If the [M] button is pressed, the application ends
074
075     jmp    main           ; Repeat
076
077 finish:                          ; ** Application End Processing **
078     jmp    _game_end     ; Application end
079
080
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
081 ; *-----*
082 ; * Displays pattern on entire LCD *
083 ; * Input c: Basic display pattern *
084 ; *-----*
085 matrix: ; **** Draws one LCD screen ****
086
087     push    acc      ; Pushes each register onto the stack
088     push    b
089     push    c
090     push    XBNK
091
092 xb0_a: mov     #000h,XBNK ; Specifies the display RAM bank address (BANK0)
093     mov     #080h,b
094
095 la1:  ld      c      ; c: Display data
096     call   line2    ; 2-line display
097     ld     b      ; Advances address two lines ahead
098     add    #010h    ;
099     st     b      ;
100    bnz    la1     ; Repeats until end of bank is reached
101
102 xb1_a: mov     #001h,XBNK ; Specifies the display RAM bank address (BANK1)
103     mov     #080h,b
104
105 la2:  ld      c      ; c: Display data
106     call   line2    ; 2-line display
107     ld     b      ; Advances address two lines ahead
108     add    #010h    ;
109     st     b      ;
110    bnz    la2     ; Repeats until end of bank is reached
111
112     pop    XBNK    ; Pops the registers from the stack
113     pop    c
114     pop    b
115     pop    acc
116
117     ret          ; Matrix end
118
119
120 line2: ; **** LCD 2-line display ****
121
122     push   acc     ; Pushes each register onto the stack
123     push   b
124     push   c
125     push   PSW
126     push   OCR
127     mov    #osc_rc,OCR ; Specifies the system clock
128     set1   PSW,1    ; Selects data RAM bank 1
129     st     c      ; Stores display data in c
130     ld     b      ; Sets the display RAM address in r2
```

```
131     st     r2     ;
132
133 lp1:           ; **** First line display processing ****
134     ld     c     ; Transfers the display data to display RAM
135     st     @r2   ;
136     inc   r2     ; Advances the address to the next display position
137     ld     r2
138     and   #00fh  ; If the display position is not at the right end of the first line...
139     xor   #006h  ;
140     bnz  lp1    ; ...repeat
141
142     ld     c     ; Inverts the bit pattern in the c register
143     xor   #0ffh  ;
144     st     c     ;
145
146 lp2:           ; **** Second line display processing
147     ld     c     ; Transfers the display data to display RAM
148     st     @r2   ;
149     inc   r2     ; Advances the address to the next display position
150     ld     r2     ;
151     and   #00fh  ; If the display position is not at the right end of the second
                        line...
152     xor   #00ch  ;
153     bnz  lp2    ; ...repeat
154
155     pop   OCR    ; Pops registers off of the stack
156     pop   PSW   ;
157     pop   c     ;
158     pop   b     ;
159     pop   acc   ;
160
161     ret                ; line2 end
162
163     end
```

# LCD Character Pattern Display

This sample program displays the text "SEGA 1998" on the LCD (XRAM).

Because the built-in fonts cannot be used from an application, the font pattern data must be prepared beforehand.

This program calls "putch", which writes the specified font pattern at the coordinates specified by registers B and C in the main routine. The font information (8 x 8 dot data) starts in line 222.

```
001; Tab width = 4
002
003;-----
004; ** LCD display processing sample 2 **
005;
006; ·Clears the display image by filling display RAM with zeroes
007; ·Displays character pattern in a specified position
008;-----
009; 1.00 981208 SEGA Enterprises,LTD.
010;-----
011
012 chip    Lc868700    ; Specifies the chip type for the assembler
013 world   external   ; External memory program
014
015 public  main       ; Symbol referenced from ghead.asm
016
017 extern  _game_end   ; Application end
018
019
020; **** Definition of System Constants ****
021
022                ; OCR (Oscillation Control Register) settings
023 osc_rc    equ 081h  ; Specifies internal RC oscillation for the system clock
024 osc_xt    equ 082h  ; Specifies crystal oscillation for the system clock
025
026
027; *** Data Segment ****
028
029          dseg       ; Data segment start
030
031 r0:      ds        1    ; Indirect addressing register r0
032 r1:      ds        1    ; Indirect addressing register r1
033 r2:      ds        1    ; Indirect addressing register r2
034 r3:      ds        1    ; Indirect addressing register r3
035          ds        12   ; Other registers reserved for the system
036
037
```



## C. Sample Program Listings

```
038 ; *** Code Segment *****
039
040   cseg           ; Code segment start
041
042 ; *-----*
043 ; * User program                               *
044 ; *-----*
045 main:
046 call    cls           ; Clears the LCD display image
047
048 mov     #1,c          ; Horizontal coordinate
049 mov     #1,b          ; Vertical coordinate
050 mov     #0ah,acc      ; Character code 'S'
051 call    putchar       ; Single character display
052
053 mov     #2,c
054 mov     #1,b
055 mov     #0bh,acc      ; 'E'
056 call    putchar
057
058 mov     #3,c
059 mov     #1,b
060 mov     #0ch,acc      ; 'G'
061 call    putchar
062
063 mov     #4,c
064 mov     #1,b
065 mov     #0dh,acc      ; 'A'
066 call    putchar
067
068 mov     #1,c
069 mov     #2,b
070 mov     #1,acc        ; '1'
071 call    putchar
072
073 mov     #2,c
074 mov     #2,b
075 mov     #9,acc        ; '9'
076 call    putchar
077
078 mov     #3,c
079 mov     #2,b
080 mov     #9,acc        ; '9'
081 call    putchar
082
083 mov     #4,c
084 mov     #2,b
085 mov     #8,acc        ; '8'
086 call    putchar
087
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
088 loop0:                ; ** [M] (mode) Button Check **
089     ld     P3
090     bn     acc,6,finish ; If the [M] button is pressed, the application ends
091
092     jmp    loop0 ; Repeat
093
094 finish:                ; ** Application End Processing **
095     jmp    _game_end; Application end
096
097
098 ; *-----*
099 ; * Clearing the LCD Display Image      *
100 ; *-----*
101 cls:
102     push   OCR          ; Pushes the OCR value onto the stack
103     mov    #osc_rc,OCR ; Specifies the system clock
104
105     mov    #0,XBNK      ; Specifies the display RAM bank address (BANK0)
106     call   cls_s        ; Clears the data in that bank
107
108     mov    #1,XBNK      ; Specifies the display RAM bank address (BANK1)
109     call   cls_s        ; Clears the data in that bank
110     pop    OCR          ; Pops the OCR value off of the stack
111
112     ret                    ; cls end
113
114 cls_s:                  ; *** Clearing One Bank of Display RAM ***
115     mov    #80h,r2      ; Points the indirect addressing register at the start
                            ; of display RAM
116     mov    #80h,b       ; Sets the number of loops in loop counter b
117 loop3:
118     mov    #0,@r2       ; Writes "0" while incrementing the address
119     inc    r2           ;
120     dbnz   b,loop3     ; Repeats until b is "0"
121
122     ret                    ; cls_s end
123
124
```

```
125 ; *-----*
126 ; * Displaying One Character in a Specified Position *
127 ; * Inputs:          acc:      Character code *
128 ; *                c:        Horizontal position of character *
129 ; *                b:        Vertical position of character *
130 ; *-----*
131  putch:
132      push  XBNK
133      push  acc
134      call  locate    ; Calculates display RAM address according to coordinates
135      pop   acc
136      call  put_chara ; Displays one character
137      pop   XBNK
138
139      ret           ; putch end
140
141
142  locate: ; **** Calculating the Display RAM Address According to the Display
           Position Specification ****
143      ; ** Inputs: c: Horizontal position (0 to 5) b: Vertical position (0 to 3)
144      ; ** Outputs: r2: RAM address XBNK: Display RAM bank
145
146      ; *** Determining the Display RAM Bank Address ***
147      ld    b           ; Jump to next1 when b >= 2
148      sub   #2          ;
149      bn    PSW,7,next1 ;
150
151      mov   #00h,XBNK   ; Specifies the display RAM bank address (BANK0)
152      br   next2
153  next1:
154      st    b
155      mov   #01h,XBNK   ; Specifies the display RAM bank address (BANK1)
156  next2:
157
158      ; *** Calculating the RAM Address for a Specified Position on the Display ***
159      ld    b           ; b * 40h + c + 80h
160      rol               ;
161      rol               ;
162      rol               ;
163      rol               ;
164      rol               ;
165      rol               ;
166      add   c           ;
167      add   #80h        ;
168      st    r2          ; Stores the RAM address in r2
169
170      ret           ; locate end
171
172
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
173 put_chara:
174     push   PSW           ; Pushes the PSW value onto the stack
175     setl   PSW,1        ; Selects data RAM bank 1
176
177                                 ; *** Calculating the Character Data Address ***
178     rol    (TRH,TRL) = acc*8 + fontdata
179     rol
180     rol
181     add    #low(fontdata) ;
182     st     TRL           ;
183     mov    #0,acc        ;
184     addc   #high(fontdata) ;
185     st     TRH           ;
186
187     push   OCR           ; Pushes the OCR value onto the stack
188     mov    #osc_rc,OCR   ; Specifies the system clock
189
190     mov    #0,b          ; Offset value for loading the character data
191     mov    #4,c          ; Loop counter
192 loop1:
193     ld     b              ; Loads the display data for the first line
194     ldc
195     inc    b              ; Increments the load data offset by 1
196     st     @r2           ; Transfers the display data to display RAM
197     ld     r2            ; Adds 6 to the display RAM address
198     add    #6            ;
199     st     r2            ;
200
201     ld     b              ; Loads the display data for the second line
202     ldc
203     inc    b              ; Increments the load data offset by 1
204     st     @r2           ; Transfers the display data to display RAM
205     ld     r2            ; Adds 10 to the display RAM address
206     add    #10           ;
207     st     r2            ;
208
209     dec    c              ; Decrements the loop counter
210     ld     c              ;
211     bnz   loop1          ; Repeats for 8 lines (four times)
212
213     pop    OCR           ; Pops the OCR value off of the stack
214     pop    PSW           ; Pops the PSW value off of the stack
215
216     ret                    ; put_chara end
217
218
```

```
219 ; *-----*
220 ; * Character Bit Image Data          *
221 ; *-----*
222 fontdata:
223   db 07ch, 0e6h, 0c6h, 0c6h, 0c6h, 0ceh, 07ch, 000h   ; '0' 00
224   db 018h, 038h, 018h, 018h, 018h, 018h, 03ch, 000h   ; '1' 01
225   db 07ch, 0c6h, 0c6h, 00ch, 038h, 060h, 0feh, 000h   ; '2' 02
226   db 07ch, 0e6h, 006h, 01ch, 006h, 0e6h, 07ch, 000h   ; '3' 03
227   db 00ch, 01ch, 03ch, 06ch, 0cch, 0feh, 00ch, 000h   ; '4' 04
228   db 0feh, 0c0h, 0fch, 006h, 006h, 0c6h, 07ch, 000h   ; '5' 05
229   db 01ch, 030h, 060h, 0fch, 0c6h, 0c6h, 07ch, 000h   ; '6' 06
230   db 0feh, 0c6h, 004h, 00ch, 018h, 018h, 038h, 000h   ; '7' 07
231   db 07ch, 0c6h, 0c6h, 07ch, 0c6h, 0c6h, 07ch, 000h   ; '8' 08
232   db 07ch, 0c6h, 0c6h, 07eh, 006h, 00ch, 078h, 000h   ; '9' 09
233
234   db 07ch, 0e6h, 076h, 038h, 0dch, 0ceh, 07ch, 000h   ; 'S' 0a
235   db 0feh, 0c0h, 0c0h, 0f8h, 0c0h, 0c0h, 0feh, 000h   ; 'E' 0b
236   db 07ch, 0e6h, 0c0h, 0dch, 0c6h, 0e6h, 07ch, 000h   ; 'G' 0c
237   db 01eh, 036h, 066h, 0c6h, 0c6h, 0feh, 0c6h, 000h   ; 'A' 0d
```

# Counter That Uses Base Timer Interrupts

This sample program detects and counts interrupts that are generated by the base timer every 0.5 seconds.

When an interrupt is generated, the program jumps to line 35 of "GHEAD.ASM", and then jumps from there to the label "INT\_1B". Base timer interrupt processing starts in line 108, but here internal clock processing is performed. After performing the clock processing in ROM once up to line 112, control jumps to the label "int\_BaseTimer" in "B\_TIMER1.ASM".

In "B\_TIMER1.ASM", the label "int\_BaseTimer", which is referenced by an external program, is declared with a PUBLIC declaration (line 16). The user's base timer interrupt handler starts from line 242. The counter is incremented within this interrupt handler.

Control returns to "GHEAD.ASM", the contents of the IE register are returned to the value that it had when "int\_1b" was called, and then control returns from the interrupt (IRET).

The counter value is always displayed on the LCD by the main routine.

### GHEAD.ASM

```
001  chip    Lc868700
002  world  external
003 ; *-----*
004 ; * External header program Ver 1.00*
005 ; * 05/20-'98*
006 ; *-----*
007
008  public          fm_wrt_ex_exit, fm_vrf_ex_exit
009  public          fm_prd_ex_exit, timer_ex_exit, _game_start, _game_end
010  other_side_symbol  fm_wrt_in, fm_vrf_in
011  other_side_symbol  fm_prd_in, timer_in, game_end
012
013  extern          main          ; Symbol in the user program
014  extern          int_BaseTimer ; Symbol in the user program
015
016 ; *-----*
017 ; * Vector table(?)*
018 ; *-----*
019  cseg
020  org 0000h
021  _game_start:
022  ;reset:
023                jmpfmain          ; main program jump
024  org 0003h
025  ;int_03:
026                jmp int_03
027  org 000bh
028  ;int_0b:
029                jmp int_0b
030  org 0013h
031  ;int_13:
032                jmp int_13
033  org 001bh
```

```
034    ;int_1b:
035        jmp int_1b
036    org 0023h
037    ;int_23:
038        jmp int_23
039    org 002bh
040    ;int_2b:
041        jmp int_2b
042    org 0033h
043    ;int_33:
044        jmp int_33
045    org 003bh
046    ;int_3b:
047        jmp int_3b
048    org 0043h
049    ;int_43:
050        jmp int_43
051    org 004bh
052    ;int_4b:
053        jmp int_4b
054    ; *-----*
055    ; * interrupt programs*
056    ; *-----*
057    int_03:
058        reti
059    int_0b:
060        reti
061    int_13:
062        reti
063    int_23:
064        reti
065    int_2b:
066        reti
067    int_33:
068        reti
069    int_3b:
070        reti
071    ; *-----*
072    int_43:
073        reti
074    int_4b:
075        clr1p3int,1; interrupt flag clear
076        reti
077
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
078  org 0100h
079  ; *-----*
080  ; * flash memory write external program*
081  ; *-----*
082  fm_wrt_ex:
083      change fm_wrt_in
084  fm_wrt_ex_exit:
085      ret
086  org 0110h
087  ; *-----*
088  ; * flash memory verify external program*
089  ; *-----*
090  fm_vrf_ex:
091      change fm_vrf_in
092  fm_vrf_ex_exit:
093      ret
094
095  org 0120h
096  ; *-----*
097  ; * flash memory page read external program*
098  ; *-----*
099  fm_prd_ex:
100      change fm_prd_in
101  fm_prd_ex_exit:
102      ret
103
104  org 0130h
105  ; *-----*
106  ; * flash memory => timer call external program *
107  ; *-----*
108  int_1b:
109  timer_ex:
110      pushie
111      clrlic,7 ; interrupt prohibition
112      changetimer_in
113  timer_ex_exit:
114      callint_BaseTimer; (User base timer interrupt processing)
115      pop ie
116      reti
117
118  org 01f0h
119  _game_end:
120      change game_end
121  end
```



**B\_TIMER1.ASM**

```
001 ; Tab width = 4
002
003 ;-----
004 ; ** Base Timer Interrupt Usage Sample 1 **
005 ;
006 ; ·Counts base timer interrupts (every 0.5 seconds)
007 ; ·Displays the counter value as a two digit decimal number on the LCD
008 ;-----
009 ; 1.00 981208 SEGA Enterprises,LTD.
010 ;-----
011
012 chip      LC868700          ; Specifies the chip type for the assembler
013 world     external         ; External memory program
014
015 public    main              ; Symbol referenced from ghead.asm
016 public    int_BaseTimer    ; Symbol referenced from ghead.asm
017
018 extern_game_end          ; Symbol reference to ghead.asm
019
020
021 ; **** Definition of System Constants *****
022
023                ; OCR (Oscillation Control Register) settings
024 osc_rcequ 081h          ; Specifies internal RC oscillation for the system clock
025 osc_xtequ 082h          ; Specifies crystal oscillation for the system clock
026
027
028 ; *** Data Segment *****
029
030     dsezz          ; Data segment start
031
032 r0:   ds      1          ; Indirect addressing register r0
033 r1:   ds      1          ; Indirect addressing register r1
034 r2:   ds      1          ; Indirect addressing register r2
035 r3:   ds      1          ; Indirect addressing register r3
036     ds      12          ; Other registers reserved for the system
037
038 counter:ds      1          ; Base timer interrupt counter
039 work1:   ds      1          ; For work (put2digit)
040
041
042 ; *** Code Segment *****
043
044     cseg          ; Code segment start
045
046 ; *-----*
047 ; * User program *
048 ; *-----*
049 main:
050     mov     #0,counter    ; Resets the counter value
051
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
052  call  cls           ; Clears the LCD display image
053
054  loop0:
055  mov   #2,c           ; Display position (horizontal)
056  mov   #1,b           ; Display position (vertical)
057  ld    counter        ; Moves the counter value to acc
058  call  put2digit      ; Displays the acc value (two digits)
059
060  setl  pcon,0         ; Waits in HALT mode until the next interrupt
061
062                      ; ** [M] (mode) Button Check **
063  ld    P3
064  bn   acc,6,finish    ; If the [M] button is pressed, the application ends
065
066  br   loop0          ; Repeat
067
068  finish:              ; ** Application End Processing **
069  jmp  _game_end      ; Application end
070
071
072 ; *-----*
073 ; * Displaying a Two-digit Value *
074 ; * Inputs: acc:   Numeric value *
075 ; *                               c: Horizontal position of character *
076 ; *                               b: Vertical position of character *
077 ; *-----*
078  put2digit:
079  push  b              ; Pushes the coordinate data onto the stack
080  push  c              ;
081  st    c              ; Calculates the tens digit and the ones digit
082  xor   a              ; ( acc = acc/10, work1 = acc mod 10 )
083  mov  #10,b          ;
084  div  c              ;
085  ld   b              ;
086  st   work1          ; Stores the ones digit in work1
087  ld   c              ;
088  pop  c              ; Pops the coordinate values into (c, b)
089  pop  b              ;
090  push b              ; Pushes the coordinates onto the stack again
091  push c              ;
092  call  putch          ; Displays the tens digit
093  ld   work1          ; Loads the ones digit
094  pop  c              ; Pops the coordinate values into (c, b)
095  pop  b              ;
096  inc  c              ; Moves the display coordinates to the right
097  call  putch          ; Displays the ones digit
098
099  ret                  ; put2digit end
100
101
```

## C. Sample Program Listings

```
102 ; *-----*
103 ; * Clearing the LCD Display Image *
104 ; *-----*
105 cls:
106     push   OCR           ; Pushes the OCR value onto the stack
107     mov    #osc_rc,OCR   ; Specifies the system clock
108
109     mov    #0,XBNK       ; Specifies the display RAM bank address (BANK0)
110     call   cls_s         ; Clears the data in that bank
111
112     mov    #1,XBNK       ; Specifies the display RAM bank address (BANK1)
113     call   cls_s         ; Clears the data in that bank
114     pop    OCR           ; Pops the OCR value off of the stack
115
116     ret                    ; cls end
117
118 cls_s: ; Clearing One Bank of Display RAM
119     mov    #80h,r2       ; Points the indirect addressing register at the start
                            ; of display RAM
120     mov    #80h,b        ; Sets the number of loops in loop counter b
121 loop3:
122     mov    #0,@r2        ; Writes "0" while incrementing the address
123     inc    r2            ;
124     dbnz  b,loop3        ; Repeats until b is "0"
125
126     ret                    ; cls_s end
127
128
129 ; *-----*
130 ; * Displaying One Character in a Specified Position*
131 ; * Inputs: acc:Character code *
132 ; *      c:Horizontal position of character*
133 ; *      b:Vertical position of character*
134 ; *-----*
135 patch:
136     push   XBNK
137     push   acc
138     call   locate        ; Calculates display RAM address according to coordinates
139     pop    acc
140     call   put_chara     ; Displays one character
141     pop    XBNK
142
143     ret                    ; patch end
144
145
146 locate: ;****Calculating the Display RAM Address According to the Display Positi
          ; on Specification****
147 ; ** Inputs: c: Horizontal position (0 to 5) b: Vertical position (0 to 3)
148 ; ** Outputs: r2: RAM address XBNK: Display RAM bank
149
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
150 ; *** Determining the Display RAM Bank Address ***
151     ld         b             ; Jump to next1 when b >= 2
152     sub        #2           ;
153     bn         PSW,7,next1   ;
154
155     mov        #00h,XBNK; Specifies the display RAM bank address (BANK0)
156     br         next2
157 next1:
158     st         b
159     mov        #01h,XBNK; Specifies the display RAM bank address (BANK1)
160 next2:
161
162 ; *** Calculating the RAM Address for a Specified Position on the Display ***
163     ld         b             ; b * 40h + c + 80h
164     rol                ;
165     rol                ;
166     rol                ;
167     rol                ;
168     rol                ;
169     rol                ;
170     add        c             ;
171     add        #80h          ;
172     st         r2           ; Stores the RAM address in r2
173
174     ret                ; locate end
175
176
177 put_chara:
178     push       PSW          ; Pushes the PSW value onto the stack
179     setl       PSW,1        ; Selects data RAM bank 1
180
181     ; *** Calculating the Character Data Address ***
182     rol                ; (TRH,TRL) = acc*8 + fontdata
183     rol                ;
184     rol                ;
185     add        #low(fontdata) ;
186     st         TRL          ;
187     mov        #0,acc       ;
188     addc       #high(fontdata) ;
189     st         TRH          ;
190
191     push       OCR          ; Pushes the OCR value onto the stack
192     mov        #osc_rc,OCR   ; Specifies the system clock
193
194     mov        #0,b         ; Offset value for loading the character data
195     mov        #4,c         ; Loop counter
196 loop1:
197     ld         b             ; Loads the display data for the first line
198     ldc                ;
199     inc        b             ; Increments the load data offset by 1
```

## C. Sample Program Listings

```
200      st    @r2          ; Transfers the display data to display RAM
201      ld    r2          ; Adds 6 to the display RAM address
202      add   #6          ;
203      st    r2          ;
204
205      ld    b           ; Loads the display data for the second line
206      ldc           ;
207      inc   b           ; Increments the load data offset by 1
208      st    @r2          ; Transfers the display data to display RAM
209      ld    r2          ; Adds 10 to the display RAM address
210      add   #10         ;
211      st    r2          ;
212
213      dec   c           ; Decrements the loop counter
214      ld    c           ;
215      bnz   loop1       ; Repeats for 8 lines (four times)
216
217      pop   OCR          ; Pops the OCR value off of the stack
218      pop   PSW         ; Pops the PSW value off of the stack
219
220      ret              ; put_chara end
221
222
223 ; *-----*
224 ; * Character Bit Image Data                *
225 ; *-----*
226 fontdata:
227 db 07ch, 0e6h, 0c6h, 0c6h, 0c6h, 0ceh, 07ch, 000h ; '0' 00
228 db 018h, 038h, 018h, 018h, 018h, 018h, 03ch, 000h ; '1' 01
229 db 07ch, 0c6h, 0c6h, 00ch, 038h, 060h, 0feh, 000h ; '2' 02
230 db 07ch, 0e6h, 006h, 01ch, 006h, 0e6h, 07ch, 000h ; '3' 03
231 db 00ch, 01ch, 03ch, 06ch, 0cch, 0feh, 00ch, 000h ; '4' 04
232 db 0feh, 0c0h, 0fch, 006h, 006h, 0c6h, 07ch, 000h ; '5' 05
233 db 01ch, 030h, 060h, 0fch, 0c6h, 0c6h, 07ch, 000h ; '6' 06
234 db 0feh, 0c6h, 004h, 00ch, 018h, 018h, 038h, 000h ; '7' 07
235 db 07ch, 0c6h, 0c6h, 07ch, 0c6h, 0c6h, 07ch, 000h ; '8' 08
236 db 07ch, 0c6h, 0c6h, 07eh, 006h, 00ch, 078h, 000h ; '9' 09
237
238
239 ; *-----*
240 ; * Base Timer Interrupt Handler            *
241 ; *-----*
242 int_BaseTimer:
243     push  acc          ; Pushes the register that was used onto the stack
244     inc   counter      ; Increments the counter
245     ld    counter      ; If the counter reaches 100...
246     bne  #100,next3    ;
247     mov  #0,counter    ; Resets the counter
248 next3:
249     pop   acc          ; Pops the register back off of the stack
250
251     ret              ; (User) interrupt processing end
```

# Button Press Detection

This sample program checks the status (pressed or not pressed) of the Visual Memory buttons (except for the reset button and the mode button), and displays on the LCD any button that was pressed.

Line 46 writes 0FFH to port 3, pulling up all bits.

The button press status is loaded in line 51 by loading the status of port 3 into the ACC. If there is a button that is being pressed at this point, the corresponding bit is reset to "0".

In line 52, the bits are checked to see if they are set (i.e., the corresponding button is not being pressed), and then control proceeds to the next button check processing. If a button is being pressed, the condition on line 52 becomes false, and the processing that is indicated for that button is performed.

---

**Note:** The port 3 interrupt is enabled immediately after this program is called from the system BIOS. Furthermore, because the port 3 interrupt is a level interrupt, the interrupt remains in effect while a button is being pressed. Disabling the port 3 interrupt improves the overall performance of applications. Note that in an application that cancels HALT mode in response to a port 3 interrupt, the port 3 interrupt must be enabled beforehand.

---

```
001 ; Tab width = 4
002
003 ;-----
004 ; ** Button Status Detection Sample 1 **
005 ;
006 ; Reads the button statuses and displays the button that is being pressed on
    the LCD
007 ;-----
008 ; 1.00 981208 SEGA Enterprises,LTD.
009 ;-----
010
011 chip    LC868700      ; Specifies the chip type for the assemble
012 world  external     ; External memory program
013
014 public  main         ; Symbol referenced from ghead.asm
015
016 extern  _game_end    ; Symbol reference to ghead.asm
017
018
019 ; **** Definition of System Constants ****
020
021 ; OCR (Oscillation Control Register) settings
022 osc_rc  equ 04dh     ; Specifies internal RC oscillation for the system clock
023 osc_xt  equ 0efh     ; Specifies crystal oscillation for the system clock
024
025
```

## C. Sample Program Listings

```
026 ; *** Data Segment *****
027
028         dseg                ; Data segment start
029
030 r0      ds          1        ; Indirect addressing register r0
031 r1      ds          1        ; Indirect addressing register r1
032 r2      ds          1        ; Indirect addressing register r2
033 r3      ds          1        ; Indirect addressing register r3
034         ds          12       ; Other registers reserved for the system
035
036 ; *** Code Segment *****
037
038         cseg                ; Code segment start
039
040 ; *-----*
041 ; * User program                      *
042 ; *-----*
043 main:
044         call   cls          ; Clears the LCD display image
045
046         mov   #0ffh,P3     ; P3 initialization (pull-up setting)
047
048 loop0:
049                                     ; ** [A] Button Check **
050         mov   #0,b
051         ld   P3            ; Loads the status of P3
052         bp   acc,4,next3   ; next3 if [A] button is being pressed
053         mov   #1,b        ; Display character code 'A'
054 next3:
055         ld   b
056         mov   #4,c        ; Display coordinate (horizontal)
057         mov   #3,b        ; Display coordinate (vertical)
058         call  putchar     ; Displays single character
059
060                                     ; ** [B] Button Check **
061         mov   #0,b
062         ld   P3
063         bp   acc,5,next4   ; next4 if [B] button is being pressed
064         mov   #2,b        ; Display character code 'B'
065 next4:
066         ld   b
067         mov   #5,c
068         mov   #2,b
069         call  putchar
070
071                                     ; ** [↑] Button Check **
072         mov   #0,b
073         ld   P3
074         bp   acc,0,next5   ; next5 if [↑] button is being pressed
075         mov   #3,b        ; Display character code '↑'
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
076 next5:
077     ld    b
078     mov   #1,c
079     mov   #1,b
080     call  putchar
081
082                                     ; ** [→] Button Check **
083     mov   #0,b
084     ld    P3
085     bp    acc,3,next6                ; next6 if [→] button is being pressed
086     mov   #4,b                        ; Display character code '→'
087 next6:
088     ld    b
089     mov   #2,c
090     mov   #2,b
091     call  putchar
092
093                                     ; ** [↓] Button Check **
094     mov   #0,b
095     ld    P3
096     bp    acc,1,next7                ; next7 if [↓] button is being pressed
097     mov   #5,b                        ; Display character code '↓'
098 next7:
099     ld    b
100     mov   #1,c
101     mov   #3,b
102     call  putchar
103
104                                     ; ** [←] Button Check **
105     mov   #0,b
106     ld    P3
107     bp    acc,2,next8                ; next8 if [←] button is being pressed
108     mov   #6,b                        ; Display character code '←'
109 next8:
110     ld    b
111     mov   #0,c
112     mov   #2,b
113     call  putchar
114
115                                     ; ** [S] Button Check **
116     mov   #0,b
117     ld    P3
118     bp    acc,7,next9                ; next9 if [S] button is being pressed
119     mov   #8,b                        ; Display character code 'S'
120 next9:
121     ld    b
122     mov   #4,c
123     mov   #1,b
124     call  putchar
125
```



## C. Sample Program Listings

```
126                                     ; ** [M] Button Check **
127     ld     P3
128     bn     acc,6,finish               ; If the [M] button is pressed, the application ends
129
130     brf    loop0                       ; Repeat
131
132 finish:                               ; ** Application End Processing **
133     jmp    _game_end                   ; Application end
134
135
136 ; *-----*
137 ; * Clearing the LCD Display Image *
138 ; *-----*
139 cls:
140     push   OCR                          ; Pushes the OCR value onto the stack
141     mov    #osc_rc,OCR                   ; Specifies the system clock
142
143     mov    #0,XBNK                       ; Specifies the display RAM bank address (BANK0)
144     call   cls_s                          ; Clears the data in that bank
145
146     mov    #1,XBNK                       ; Specifies the display RAM bank address (BANK1)
147     call   cls_s                          ; Clears the data in that bank
148     pop    OCR                          ; Pops the OCR value off of the stack
149
150     ret                                    ; cls end
151
152     cls_s:                               ; *** Clearing One Bank of Display RAM ***
153     mov    #80h,r2                       ; Points the indirect addressing register at the
154                                     ; start of display RAM
155     mov    #80h,b                         ; Sets the number of loops in loop counter b
156 loop3:
157     mov    #0,@r2                         ; Writes "0" while incrementing the address
158     inc    r2                             ;
159     dbnz   b,loop3                       ; Repeats until b is "0"
160     ret                                    ; cls_s end
161
162
163 ; *-----*
164 ; * Displaying One Character in a Specified Position*
165 ; * Inputs: acc:Character code *
166 ; *           c:   Horizontal position of character *
167 ; *           b:   Vertical position of character *
168 ; *-----*
169 patch:
170     push   XBNK
171     push   acc
172     call   locate                         ; Calculates display RAM address according to coordinates
173     pop    acc
174     call   put_chara                      ; Displays one character
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
175         pop     XBNK
176
177         ret                ; patch end
178
179
180 locate: ; **** Calculating the Display RAM Address According to the Display
           Position Specification ****
181         ; ** Inputs: c: Horizontal position (0 to 5) b: Vertical position (0 to 3)
182         ; ** Outputs: r2: RAM address XBNK: Display RAM bank
183
184         ; *** Determining the Display RAM Bank Address ***
185         ld      b                ; Jump to next1 when b >= 2
186         sub    #2                ;
187         bn     PSW,7,next1;
188
189         mov    #00h,XBNK; Specifies the display RAM bank address (BANK0)
190         br    next2
191 next1:
192         st     b
193         mov    #01h,XBNK; Specifies the display RAM bank address (BANK1)
194 next2:
195
196         ; *** Calculating the RAM Address for a Specified
           Position on the Display ***
197         ld     b                ; b * 40h + c + 80h
198         rol                    ;
199         rol                    ;
200         rol                    ;
201         rol                    ;
202         rol                    ;
203         rol                    ;
204         add    c                ;
205         add    #80h            ;
206         st     r2              ; Stores the RAM address in r2
207
208         ret                ; locate end
209
210
211 put_chara:
212         push   PSW              ; Pushes the PSW value onto the stack
213         set1   PSW,1            ; Selects data RAM bank 1
214
215         ; *** Calculating the Character Data Address ***
216         rol                    ; (TRH,TRL) = acc*8 + fontdata
217         rol                    ;
218         rol                    ;
219         add    #low(fontdata)  ;
220         st     TRL             ;
221         mov    #0,acc          ;
222         addc   #high(fontdata) ;
```

## C. Sample Program Listings

```
223         st          TRH          ;
224
225         push   OCR          ; Pushes the OCR value onto the stack
226         mov     #osc_rc,OCR   ; Specifies the system clock
227
228         mov     #0,b         ; Offset value for loading the character data
229         mov     #4,c         ; Loop counter
230 loop1:
231         ld      b            ; Loads the display data for the first line
232         ldc     ldc          ;
233         inc     b            ; Increments the load data offset by 1
234         st     @r2          ; Transfers the display data to display RAM
235         ld     r2           ; Adds 6 to the display RAM address
236         add    #6           ;
237         st     r2           ;
238
239         ld     b            ; Loads the display data for the second line
240         ldc     ldc          ;
241         inc     b            ; Increments the load data offset by 1
242         st     @r2          ; Transfers the display data to display RAM
243         ld     r2           ; Adds 10 to the display RAM address
244         add    #10          ;
245         st     r2           ;
246
247         dec    c            ; Decrements the loop counter
248         ld     c            ;
249         bnz    loop1        ; Repeats for 8 lines (four times)
250
251         pop    OCR          ; Pops the OCR value off of the stack
252         pop    PSW          ; Pops the PSW value off of the stack
253
254         ret          ; put_chara end
255
256
257 ; *-----*
258 ; * Character Bit Image Data *
259 ; *-----*
260 fontdata:
261 db 000h, 000h, 038h, 038h, 038h, 000h, 000h, 000h ; '•' 00
262 db 01eh, 036h, 066h, 0c6h, 0c6h, 0feh, 0c6h, 000h ; 'A' 01
263 db 0fch, 066h, 066h, 07ch, 066h, 066h, 0fch, 000h ; 'B' 02
264
265 db 010h, 038h, 07ch, 0feh, 038h, 038h, 038h, 000h ; '↑' 03
266 db 010h, 018h, 0fch, 0feh, 0fch, 018h, 010h, 000h ; '→' 04
267 db 038h, 038h, 038h, 0feh, 07ch, 038h, 010h, 000h ; '↓' 05
268 db 010h, 030h, 07eh, 0feh, 07eh, 030h, 010h, 000h ; '←' 06
269
270 db 0c6h, 0eeh, 0feh, 0d6h, 0c6h, 0c6h, 0c6h, 000h ; 'M' 07
271 db 07ch, 0e6h, 076h, 038h, 0dch, 0ceh, 07ch, 000h ; 'S' 08
```

# Using the PWM Sound Source

This sample program alternately generates a high tone (781Hz) and a low tone (342Hz).

The important portion of this sample is the subroutine that starts from line 72.

"SndInit" readies the program to use PWM. "Snd1(2)Start" sets the frequency that is generated by timer 1, and begins the counting operation. "SndStop" stops the counting operation and stops the audio output.

```
001 ; Tab width = 4
002
003 ;-----
004 ; ** Sound Usage Sample 1 **
005 ;
006 ; ·Intermittently outputs two tones (high/low)
007 ; (Low tone for 0.5 seconds - Silence for 0.5 seconds - High tone for 0.5
    seconds - Silence for 0.5 seconds...)
008 ;-----
009 ; 1.00 981208 SEGA Enterprises,LTD.
010 ;-----
011
012 chip    LC868700          ; Specifies the chip type for the assembler
013 world   external        ; External memory program
014
015 public  main             ; Symbol referenced from ghead.asm
016
017 extern  _game_end        ; Application end
018
019
020 ; **** Definition of System Constants ****
021
022                ; OCR (Oscillation Control Register) settings
023 osc_rc    equ 081h        ; Specifies internal RC oscillation for the system clock
024 osc_xt    equ 082h        ; Specifies crystal oscillation for the system clock
025
026
027 ; *** Data Segment ****
028
029         dseg              ; Data segment start
030
031 r0:      ds               1    ; Indirect addressing register r0
032 r1:      ds               1    ; Indirect addressing register r1
033 r2:      ds               1    ; Indirect addressing register r2
034 r3:      ds               1    ; Indirect addressing register r3
035         ds               12   ; Other registers reserved for the system
036
037
038 ; *** Code Segment ****
039
040         cseg              ; Code segment start
041
042 ; *-----*
043 ; * User program *
044 ; *-----*
```

## C. Sample Program Listings

```
045 main:
046 call    SndInit          ; Sound initialization
047
048 loop0:
049     call Snd1Start       ; Starts generating tone at approximately 342Hz
050     setl PCON,0         ; HALT mode until base timer interrupt (0.5 seconds)
051
052     call SndStop        ; Buzzer sound off
053     setl PCON,0         ; HALT mode until base timer interrupt (0.5 seconds)
054
055     call Snd2Start       ; Starts generating tone at approximately 781
056     setl PCON,0         ; HALT mode until base timer interrupt (0.5 seconds)
057
058     call SndStop        ; Buzzer sound off
059     setl PCON,0         ; HALT mode until base timer interrupt (0.5 seconds)
060
061
062                                     ; ** [M] (mode) Button Check **
063     ld    P3
064     bn   acc,6,finish    ; If the [M] button is pressed, the application ends
065
066     br   loop0          ; Repeat
067
068 finish:
069     jmp  _game_end      ; Application end
070
071
072 ; *-----*
073 ; * Sound Output-related Routines *
074 ; *-----*
075 SndInit:
076     clr1 P1,7          ; *** Sound Output Hardware Initialization ***
077
078     ret
079
080 Snd1Start:
081     mov  #0f0h,T1LR     ; *** Start of 342Hz Tone ***
082     mov  #0f8h,T1LC     ; Cycles = 100h - 0f0h = 16
083     mov  #0D0h,T1CNT    ; L level width = 100h - 0f8h = 8
084     mov  #0D0h,T1CNT    ; Sound output start
085
086     ret
087
088 Snd2Start:
089     mov  #0f9h,T1LR     ; *** Start of 781Hz Tone ***
090     mov  #0fch,T1LC     ; Cycles = 100h - 0f9h = 7
091     mov  #0D0h,T1CNT    ; L level width = 100h - 0fch = 4
092     mov  #0D0h,T1CNT    ; Sound output start
093
094     ret
095
096 SndStop:
097     mov  #0,T1CNT      ; *** Sound Stop ***
098     mov  #0,T1CNT      ; Stops sound output
099
100     ret
```

# Interrupt Using Timer 0

This sample program generates an interrupt once every second. The program generates a sound when the interrupt is generated.

The important portion of this program is the routine "T0Mode2Init" in lines 78 through 93. This program uses Timer 0 in mode 2, as a 16-bit counter with prescaler. Because a 32kHz signal is input to the timer, the program sets up the timer so that the signal causes an overflow in the prescaler and counter approximately every second.

Because Timer 0 generates an interrupt in response to the overflow, the program provides a handler for that interrupt. This handler increments the count, resets the interrupt source flag to "0", and then terminates interrupt processing.

The main routine determines whether the counter value is an even or odd number, and uses this information to output a high tone and a low tone in alternation. In line 61, the CPU is put into HALT mode, and operation stops until an interrupt is generated. If a timer 0 interrupt or a port 3 interrupt is generated, processing resumes from the next line.

### • GHEAD.ASM

```
001 chip      LC868700
002 world     external
003 ; *-----*
004 ; * External header program Ver 1.00*
005 ; *                               05/20-'98*
006 ; *-----*
007
008 public    fm_wrt_ex_exit, fm_vrf_ex_exit
009 public    fm_prd_ex_exit, timer_ex_exit, _game_start, _game_end
010 other_side_symbol fm_wrt_in, fm_vrf_in
011 other_side_symbol fm_prd_in, timer_in, game_end
012
013 extern    main                ; Symbol in the user program
014 extern    int_T0H             ; Symbol in the user program
015
016 ; *-----*
017 ; * Vector table(?)          *
018 ; *-----*
019 cseg
020 org 0000h
021 _game_start:
022 ;reset:
023         jmpf    main      ; main program jump
024 org 0003h
025 ;int_03:
026         jmp     int_03
027 org 000bh
028 ;int_0b:
029         jmp     int_0b
030 org 0013h
031 ;int_13:
032         jmp     int_13
033 org 001bh
```

```
034 ;int_1b:
035     jmp     int_1b
036 org 0023h
037 ;int_23:
038     jmp     int_23
039 org 002bh
040 ;int_2b:
041     jmp     int_2b
042 org 0033h
043 ;int_33:
044     jmp     int_33
045 org 003bh
046 ;int_3b:
047     jmp     int_3b
048 org 0043h
049 ;int_43:
050     jmp     int_43
051 org 004bh
052 ;int_4b:
053     jmp     int_4b
054 ; *-----*
055 ; * interrupt programs      *
056 ; *-----*
057 int_03:
058     reti
059 int_0b:
060     reti
061 int_13:
062     reti
063 int_23:
064     jmp     int_T0H      ; (To user interrupt processing)
065 int_2b:
066     reti
067 int_33:
068     reti
069 int_3b:
070     reti
071 ; *-----*
072 int_43:
073     reti
074 int_4b:
075     clr1   p3int,1      ; interrupt flag clear
076     reti
077
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
078 org 0100h
079 ; *-----*
080 ; * flash memory write external program*
081 ; *-----*
082 fm_wrt_ex:
083         change fm_wrt_in
084         fm_wrt_ex_exit:
085         ret
086 org 0110h
087 ; *-----*
088 ; * flash memory verify external program*
089 ; *-----*
090 fm_vrf_ex:
091         change fm_vrf_in
092         fm_vrf_ex_exit:
093         ret
094
095 org 0120h
096 ; *-----*
097 ; * flash memory page read external program*
098 ; *-----*
099 fm_prd_ex:
100         change fm_prd_in
101         fm_prd_ex_exit:
102         ret
103
104 org 0130h
105 ; *-----*
106 ; * flash memory => timer call external program*
107 ; *-----*
108 int_1b:
109 timer_ex:
110         push    ie
111         cml    ie,7           ; interrupt prohibition
112         change timer_in
113         timer_ex_exit:
114         pop     ie
115         reti
116
117 org 01f0h
118 _game_end:
119         change game_end
120 end
```



**• TIMER1.ASM**

```
001 ; Tab width = 4
002
003 ;-----
004 ; ** Timer/Counter T0 Interrupt Usage Sample 1 **
005 ;
006 ; Intermittently sounds the buzzer (every two seconds)
007 ;-----
008 ; 1.00 981208 SEGA Enterprises,LTD.
009 ;-----
010
011 chip    LC868700          ; Specifies the chip type for the assembler
012 world   external        ; External memory program
013
014 public  main             ; Symbol referenced from ghead.asm
015 public  int_T0H         ; Symbol referenced from ghead.asm
016
017 extern  _game_end       ; Application end
018
019
020 ; **** Definition of System Constants ****
021
022                ; OCR (Oscillation Control Register) settings
023 osc_rc      equ 04dh     ; Specifies internal RC oscillation for the system clock
024 osc_xt      equ 0efh     ; Specifies crystal oscillation for the system clock
025
026
027 ; *** Data Segment ****
028
029                dseg          ; Data segment start
030
031 r0:         ds           1    ; Indirect addressing register r0
032 r1:         ds           1    ; Indirect addressing register r1
033 r2:         ds           1    ; Indirect addressing register r2
034 r3:         ds           1    ; Indirect addressing register r3
035            ds           12    ; Other registers reserved for the system
036
037 counter: ds   1          ; Timer interrupt counter
038
039
040 ; *** Code Segment ****
041
042                cseg          ; Code segment start
043
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
044 ; *-----*
045 ; * User program *
046 ; *-----*
047 main:
048     call  SndInit           ; Sound output initialization
049     call  T0Mode2Init      ; Timer T0 initialization
050     mov   #0,counter       ; Clears counter
051
052 loop0:
053     ld    counter          ; Loads the counter value
054     bp   acc,1,next1      ; next1 if bit 0 of the counter is "1"
055
056     call  Snd2Start        ; Starts sound
057     br   next2
058 next1:
059     call  SndStop          ; Stops sound
060 next2:
061     set1  PCON,0           ; HALT mode until next interrupt
062
063
064                                     ; ** [M] (mode) Button Check **
065     ld    P3
066     bn   acc,6,finish     ; If the [M] button is pressed, the application ends
067
068     br   loop0            ; Repeat
069
070 finish:
071     jmp   _game_end       ; Application end
072
073
074 ; *-----*
075 ; * Timer/Counter T0 Initialization *
076 ; * Applied an interrupt about once per second in mode 2 (16-bit counter)*
077 ; *-----*
078 T0Mode2Init:
079     mov   #255,T0PRR      ; Sets the prescaler value
080                                     ; Since this is an 8-bit prescaler:
081                                     ; Cycle = (256-255) * 0.000183 = 0.000183 (sec.)
082     mov   #high(65536-5464),T0HR; Sets preset value (H)
083     mov   #low(65536-5464),T0LR ; Sets preset value (L)
084                                     ; As a set with the prescaler:
085                                     ; 0.000183 * 5464 = 0.999912 (^1sec)
086                                     ; An overflow occurs about once per second
087     mov   #0ffh,T0L       ; Sets up an immediate initial overflow
088     mov   #0ffh,T0H       ;
089     mov   #0e4h,T0CNT     ; Mode 2 (16-bit counter)
090                                     ; Generates an interrupt according to
091                                     ; the T0H overflow
092     ; T0 operation start
```

## C. Sample Program Listings

```
093             ret                      ; T0Mode2Init end
094
095
096 TOHStop:                ; *** T0H timer stop ***
097
098     clr1   T0CNT,7        ; T0H count operation stop
099     ret
100
101
102 ; *-----*
103 ; * Timer T0H Interrupt Handler *
104 ; *-----*
105 int_T0H:                ; *** T0H Interrupt Handler ***
106     inc    counter
107
108     clr1   T0CNT,3        ; Clears the timer T0H interrupt source
109     reti
110
111
112 ; *-----*
113 ; * Sound Output-related Routines *
114 ; *-----*
115 SndInit:                ; *** Sound Output Hardware Initialization ***
116     clr1   P1,7          ; Sets the sound output port
117
118     ret
119
120 Snd1Start:              ; *** Start of 342Hz Tone ***
121     mov    #0f0h,T1LR    ; Cycles = 100h - 0f0h = 16
122     mov    #0f8h,T1LC    ; L level width = 100h - 0f8h = 8
123     mov    #0D4h,T1CNT   ; Sound output start
124
125     ret
126
127 Snd2Start:              ; *** Start of 781Hz Tone ***
128     mov    #0f9h,T1LR    ; Cycles = 100h - 0f9h = 7
129     mov    #0fch,T1LC    ; L level width = 100h - 0fch = 4
130     mov    #0D4h,T1CNT   ; Sound output start
131
132     ret
133
134 SndStop:                ; *** Sound Stop ***
135     mov    #0,T1CNT      ; Stops sound output
136
137     ret
```

# Serial Communications (Sending Side)

This sample program uses the serial interface to send data values from 0 to 99.

---

**Caution:** Conduct serial communications with crystal oscillation.

---

Perform reception with the "Serial Communications (Receiving Side)" that is described on the following page.

Line 53 disables automatic low battery detection. The actual routines are "BattChkOn" and "BattChkOff" in line 158 and beyond.

Line 56 initializes the serial interface. The actual initialization routine starts in line 95. This routine is very detailed, so we recommend simply copying this code and using it as is.

After initialization, a counter is incremented by the base timer interrupt, which is generated every 0.5 seconds. The value of this counter is sent through the serial interface.

If this program is halted by pressing the MODE button, the standard value is written for the serial interface again (in the "SioEnd" routine that starts from line 126), automatic low battery detection is enabled again, and the program ends.

### • GHEAD.ASM

```
001 chip      LC868700
002 world     external
003 ; *-----*
004 ; * External header program Ver 1.00*
005 ; *                               05/20-'98*
006 ; *-----*
007
008 public    fm_wrt_ex_exit, fm_vrf_ex_exit
009 public    fm_prd_ex_exit, timer_ex_exit, _game_start, _game_end
010 other_side_symbol fm_wrt_in, fm_vrf_in
011 other_side_symbol fm_prd_in, timer_in, game_end
012
013 extern    main                ; Symbol in the user program
014 extern    int_BaseTimer      ; Symbol in the user program
015
016 ; *-----*
017 ; * Vector table(?)          *
018 ; *-----*
019 cseg
020 org 0000h
021 _game_start:
022 ;reset:
023         jmpf    main    ; main program jump
024 org 0003h
025 ;int_03:
026         jmp     int_03
027 org 000bh
028 ;int_0b:
029         jmp     int_0b
```

```
030 org 0013h
031 ;int_13:
032     jmp     int_13
033 org 001bh
034 ;int_1b:
035     jmp     int_1b
036 org 0023h
037 ;int_23:
038     jmp     int_23
039 org 002bh
040 ;int_2b:
041     jmp     int_2b
042 org 0033h
043 ;int_33:
044     jmp     int_33
045 org 003bh
046 ;int_3b:
047     jmp     int_3b
048 org 0043h
049 ;int_43:
050     jmp     int_43
051 org 004bh
052 ;int_4b:
053     jmp     int_4b
054 ; *-----*
055 ; * interrupt programs      *
056 ; *-----*
057 int_03:
058     reti
059 int_0b:
060     reti
061 int_13:
062     reti
063 int_23:
064     reti
065 int_2b:
066     reti
067 int_33:
068     reti
069 int_3b:
070     reti
071 ; *-----*
072 int_43:
073     reti
074 int_4b:
075     clr1   p3int,1      ; interrupt flag clear
076     reti
077
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
078 org 0100h
079 ; *-----*
080 ; * flash memory write external program*
081 ; *-----*
082 fm_wrt_ex:
083         change fm_wrt_in
084         fm_wrt_ex_exit:
085         ret
086 org 0110h
087 ; *-----*
088 ; * flash memory verify external program*
089 ; *-----*
090 fm_vrf_ex:
091         change fm_vrf_in
092         fm_vrf_ex_exit:
093         ret
094
095 org 0120h
096 ; *-----*
097 ; * flash memory page read external program*
098 ; *-----*
099 fm_prd_ex:
100        change fm_prd_in
101        fm_prd_ex_exit:
102        ret
103
104 org 0130h
105 ; *-----*
106 ; * flash memory => timer call external program*
107 ; *-----*
108 int_1b:
109 timer_ex:
110         push  ie
111         clr1  ie,7           ; interrupt prohibition
112         change timer_in
113         timer_ex_exit:
114         call  int_BaseTimer ; User interrupt processing
115         pop  ie
116         reti
117
118 org 01f0h
119 _game_end:
120         change game_end
121 end
```

**• TIMER1.ASM**

```
001 ; Tab width = 4
002
003 ;-----
004 ; ** Serial Communications Sample 1 (Data Transmission) **
005 ;
006 ; .Sends simple data through the serial communications port on a regular cycle
007 ;-----
008 ; 1.01 990208 SEGA Enterprises,LTD.
009 ;-----
010
011 chip    LC868700          ; Specifies the chip type for the assembler
012 world   external        ; External memory program
013
014 public  main             ; Symbol referenced from ghead.asm
015 public  int_BaseTimer   ; Symbol referenced from ghead.asm
016
017 extern  _game_end       ; Application end
018
019
020 ; **** Definition of System Constants *****
021
022                ; OCR (Oscillation Control Register) settings
023 osc_rc    equ 04dh       ; Specifies internal RC oscillation for the system clock
024 osc_xt    equ 0efh       ; Specifies crystal oscillation for the system clock
025
026 LowBattChkequ 06eh      ; Low battery detection flag (RAM bank 0)
027
028
029 ; *** Data Segment *****
030
031          dseg           ; Data segment start
032
033 r0:ds    1              ; Indirect addressing register r0
034 r1:ds    1              ; Indirect addressing register r1
035 r2:ds    1              ; Indirect addressing register r2
036 r3:ds    1              ; Indirect addressing register r3
037 ds      12             ; Other registers
038
039 counter: ds    1        ; Counter
040
041
042 ; *** Code Segment *****
043
044          cseg           ; Code segment start
045
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
046 ; *-----*
047 ; * User program *
048 ; *-----*
049 main:
050     push   PSW           ; Pushes the PSW value onto the stack
051     setl   PSW,1         ; Selects data RAM bank 1
052
053     call   BattChkOff    ; Turns off the low battery automatic detection function
054
055 cwait:
056     call   SioInit       ; Serial communications initialization
057     bz     start         ; Starts if VM is connected
058
059     ld     P3            ; [M] button check
060     bn     acc,6,finish  ; If the [M] button is pressed, the application ends
061
062     jmp    cwait         ; Waits until VM is connected
063 start:
064
065     setl   pcon,0        ; Waits in HALT mode until next interrupt (0.5 seconds)
066
067     mov    #0,counter    ; Resets the counter value to "0"
068 loop0:
069     ld     counter       ; Loads the counter value
070
071     call   SioSend1      ; Sends one byte
072
073     setl   pcon,0        ; Waits in HALT mode until next interrupt (0.5 seconds)
074
075                                     ; [M] (mode) Button Check
076     ld     P3
077     bn     acc,6,finish  ; If the [M] button is pressed, the application ends
078
079     jmp    loop0         ; Repeat
080
081 finish:
082                                     ; ** Application End Processing **
083     call   SioEnd        ; Serial communications end processing
084     call   BattChkOn     ; Turns on the low battery automatic detection function
085     pop    PSW           ; Pops the PSW value off of the stack
086     jmp    _game_end     ; Application end
```



## C. Sample Program Listings

```
087 ; *-----*
088 ; * Serial Communications Initialization *
089 ; * Outputs:acc = 0 : Normal end *
090 ; *          acc = 0ffh: VM not connected *
091 ; *-----*
092 ; Serial communications initialization
093 ; This sample assumes that the system clock is in crystal mode.
094
095 SioInit:
096                                     ; **** VM Connection Check ****
097     ld     P7                        ; Checks the connection status
098     and    #%00001101                ; Checks P70, P72, P73
099     sub    #%00001000                ; P70 = 0, P72 = 0, P73 = 1
100     bz     next2                     ; To next2 if connected
101
102     mov    #0ffh,acc; If not connected, abnormal end with acc = 0ffh
103     ret                                     ; SioInit end
104 next2:
105
106                                     ; **** Serial Communications Initialization ****
107     mov    #0,SCON0                  ; Specifies output as 'LSB first'
108     mov    #0,SCON1                  ; Specifies input as 'LSB first'
109     mov    #0ddh,SBR                  ; Sets the transfer rate
110     clr1   P1,0                       ; Clears the P10 latch (P10/S00)
111     clr1   P1,2                       ; Clears the P12 latch (P12/SCK0)
112     clr1   P1,3                       ; Clears the P13 latch (P13/S01)
113
114     mov    #%00000101,P1FCR           ; Sets the pin functions
115     mov    #%00000101,P1DDR           ; Sets the pin functions
116
117     mov    #0,SBUF0                   ; Clears the transfer buffer
118     mov    #0,SBUF1                   ; Clears the transfer buffer
119
120     ret                                     ; SioInit end
121
122
123 ; *-----*
124 ; * Serial Communications End *
125 ; *-----*
126 SioEnd:                                ; **** Serial Communications End Processing ****
127
128     mov    #0,SCON0                  ; SCON0 = 0
129     mov    #0,SCON1                  ; SCON1 = 0
130     mov    #0bfh,P1FCR               ; P1FCR = 0bfh
131     mov    #0a4h,P1DDR               ; P1DDR = 0a4h
132
133     ret                                     ; SioEnd end
134
135
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
136 ; *-----*
137 ; * Sending 1 Byte from a Serial Port *
138 ; * Inputs: acc: Transmission data *
139 ; *-----*
140 SioSend1: ; **** Sending 1 Byte ****
141
142     push    acc ; Pushes the transmission data onto the stack
143
144     sslpl:  ld     SCON0 ; Waits, if the previous data is still being sent
145             bp     acc,3,sslpl ;
146
147             pop    acc ; Pops the transmission data off of the stack
148
149             st     SBUF0 ; Sets the data to be transferred
150             set1   SCON0,3 ; Starts sending
151
152             ret    ; SioSend1 end
153
154
155 ; *-----*
156 ; * Low Battery Automatic Detection Function ON*
157 ; *-----*
158 BattChkOn:
159     push    PSW ; Pushes the PSW value onto the stack
160
161     clr1    PSW,1 ; Selects data RAM bank 0
162     mov     #0,acc ; Detects low battery (0)
163     st     LowBattChk ; Low battery automatic detection flag (RAM bank 0)
164
165     pop    PSW ; Pops the PSW value off of the stack
166     ret    ; BattChkOn end
167
168
169 ; *-----*
170 ; * Low Battery Automatic Detection Function OFF*
171 ; *-----*
172 BattChkOff:
173     push    PSW ; Pushes the PSW value onto the stack
174
175     clr1    PSW,1 ; Selects data RAM bank 0
176     mov     #0ffh,acc ; Does not detect low battery (0ffh)
177     st     LowBattChk ; Low battery automatic detection flag (RAM bank 0)
178
179     pop    PSW ; Pops the PSW value off of the stack
180     ret    ; BattChkOff end
181
182
```

```
183 ; *-----*
184 ; * Base Timer Interrupt Handler *
185 ; *-----*
186 int_BaseTimer:
187     push    PSW          ; Pushes the PSW value onto the stack
188     push    acc
189
190     setl    PSW,1        ; Selects data RAM bank 1
191
192     inc     counter      ; Increments the counter
193
194     ld      counter      ; If the counter value is:
195     bne     #100,next1   ; not 100, then next1
196     mov     #0,counter   ; 100, then reset to '0'
197 next1:
198     pop     acc
199     pop     PSW          ; Pops the PSW value off of the stack
200
201     clrl   BTCR,1        ; Clears the base timer interrupt source
202     ret                                ; User interrupt processing end
```

# Serial Communications (Receiving Side)

This sample program uses the serial interface to receive data.

Because this program is intended to primarily explain serial communications, it does not use SIO interrupts. For details on serial communications in actual practice, refer to the "General-purpose Serial Driver," described on the next page.

The program stops automatic low battery detection and initializes the serial interface.

Line 64 checks whether there is a byte of data in the serial interface. If there is, the received data is converted to a decimal value by the "put2digit" routine and is displayed on the LCD.

If this program is halted by pressing the MODE button, the standard value is written for the serial interface again (in the "SioEnd" routine that starts from line 121), automatic low battery detection is enabled again, and the program ends.

---

**Caution:** If data is sent before reception processing is completed, a data overflow occurs. This is not a problem in this sample program because the "Serial Communications (Sending Side)" sample program sends data every 0.5 seconds.  
When receiving data consecutively, use the SIO interrupts.

---

```
001 ; Tab width = 4
002
003 ;-----
004 ; ** Serial Communications Sample 2 (Data Reception) **
005 ;
006 ; ·Displays a numeric value that was received from the serial communications
    port on the LCD
007 ;-----
008 ; 1.01 990208 SEGA Enterprises,LTD.
009 ;-----
010
011 chip    LC868700          ; Specifies the chip type for the assembler
012 world  external         ; External memory program
013
014 public main              ; Symbol referenced from ghead.asm
015
016 extern _game_end         ; Application end
017
018
019 ; **** Definition of System Constants ****
020
021                ; OCR (Oscillation Control Register) settings
022 osc_rc  equ 04dh         ; Specifies internal RC oscillation for the system clock
023 osc_xt  equ 0efh         ; Specifies crystal oscillation for the system clock
024
025 LowBattChkequ 06eh       ; Low battery detection flag (RAM bank 0)
026
027
028 ; *** Data Segment ****
029
```

## C. Sample Program Listings

```
030         dseg                ; Data segment start
031
032 r0:      ds          1        ; Indirect addressing register r0
033 r1:      ds          1        ; Indirect addressing register r1
034 r2:      ds          1        ; Indirect addressing register r2
035 r3:      ds          1        ; Indirect addressing register r3
036         ds          12       ; Other registers
037
038 counter: ds          1        ; Counter
039 work1:   ds          1        ; Work (used in put2digit)
040
041
042 ; *** Code Segment *****
043
044         cseg                ; Code segment start
045
046 ; *-----*
047 ; * User program *
048 ; *-----*
049 main:
050         call   cls          ; Clears the LCD display
051         call   BattChkOff   ; Turns off the low battery automatic detection function
052
053 cwait:
054         call   SioInit      ; Serial communications initialization
055         bz     start        ; Starts if VM is connected
056
057         ld     P3           ; [M] button check
058         bn     acc,6,finish ; If the [M] button is pressed, the application ends
059
060         jmp    cwait        ; Waits until VM is connected
061 start:
062
063 loop0:
064         call   SioRecv1     ; Receives one byte
065         bnz   next4        ; If there is no received data, then goes to next4
066
067         ld     b            ; Loads the received data into acc
068         mov   #2,c         ; Display coordinates (horizontal)
069         mov   #1,b         ; Display coordinates (vertical)
070         call   put2digit    ; Displays the two-digit value on the LCD
071
072 next4:                ; ** [M] (mode) Button Check **
073         ld     P3
074         bn     acc,6,finish ; If the [M] button is pressed, the application ends
075
076         jmp    loop0       ; Repeat
077
078 finish:                ; ** Application End Processing **
079         call   SioEnd       ; Serial communications end processing
080         call   BattChkOn    ; Turns on the low battery automatic detection function
081         jmp    _game_end    ; Application end
082
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
083 ; *-----*
084 ; * Serial Communications Initialization *
085 ; * Outputs:acc = 0 : Normal end *
086 ; *          acc = 0ffh: VM not connected *
087 ; *-----*
088 ; Serial communications initialization
089 ; This sample assumes that the system clock is in crystal mode.
090 SioInit:
091             ; **** VM Connection Check ****
092     ld      P7          ; Checks the connection status
093     and     #%00001101  ; Checks P70, P72, P73
094     sub     #%00001000  ; P70 = 0, P72 = 0, P73 = 1
095     bz     next3       ; To next3 if connected
096
097     mov     #0ffh,acc   ; If not connected, abnormal end with acc = 0ffh
098     ret
099 next3:
100
101             ; **** Serial Communications Initialization ****
102     mov     #0,SCON0    ; Specifies output as 'LSB first'
103     mov     #0,SCON1    ; Specifies input as 'LSB first'
104     mov     #088h,SBR   ; Sets the transfer rate
105     clr1   P1,0         ; Clears the P10 latch (P10/S00)
106     clr1   P1,2         ; Clears the P12 latch (P12/SCK0)
107     clr1   P1,3         ; Clears the P13 latch (P13/S01)
108
109     mov     #%00000101,P1FCR ; Sets the pin functions
110     mov     #%00000101,P1DDR ; Sets the pin functions
111
112     mov     #0,SBUF0     ; Clears the transfer buffer
113     mov     #0,SBUF1     ; Clears the transfer buffer
114
115     ret
116
117
118 ; *-----*
119 ; * Serial Communications End *
120 ; *-----*
121 SioEnd:             ; **** Serial Communications End Processing ****
122
123     mov     #0,SCON0    ; SCON0 = 0
124     mov     #0,SCON1    ; SCON1 = 0
125     mov     #0bfh,P1FCR ; P1FCR = 0bfh
126     mov     #0a4h,P1DDR ; P1DDR = 0a4h
127
128     ret
129
130
```

## C. Sample Program Listings

```
131 ; *-----*
132 ; * Receiving 1 Byte from a Serial port *
133 ; * Outputs: b: Received data *
134 ; *          acc = 0 : Received data found *
135 ; *          acc = 0ffh: Received data not found*
136 ; *-----*
137 SioRecv1: ; **** Receiving 1 Byte ****
138     ld     SCON1
139     bp     acc,1,next5 ; If received data is found, then go to next5
140     bp     acc,3,next6 ; If transfer is currently in progress, then go to next6
141
142     set1   SCON1,3 ; Starts transfer
143 next6:
144     mov    #0ffh,acc; Returns with acc = 0ffh (received data not found)
145     ret    ; SioRecv1 end
146 next5:
147
148     ld     SBUF1 ; Loads the received data
149     st     b ; Copies the data into b
150
151     clr1   SCON1,1 ; Resets the transfer end flag
152
153     mov    #0,acc ; Returns with acc = 0 (received data found)
154     ret    ; SioRecv1 end
155
156
157 ; *-----*
158 ; * Displaying a two-digit value *
159 ; * Inputs: acc: Numeric value *
160 ; *          c: Horizontal position of character*
161 ; *          b: Vertical position of character*
162 ; *-----*
163 put2digit:
164     push   b ; Pushes the coordinate data onto the stack
165     push   c ;
166     st     c ; Calculates the tens digit and the ones digit
167     xor    a ; ( acc = acc/10, work1 = acc mod 10 )
168     mov    #10,b ;
169     div    ;
170     ld     b ;
171     st     work1 ; Stores the ones digit in work1
172     ld     c ;
173     pop    c ; Pops the coordinate values into (c, b)
174     pop    b ;
175     push   b ; Pushes the coordinates onto the stack again
176     push   c ;
177     call   putch ; Displays the tens digit
178     ld     work1 ; Loads the ones digit
179     pop    c ; Pops the coordinate values into (c, b)
180     pop    b ;
181     inc    c ; Moves the display coordinates to the right
182     call   putch ; Displays the ones digit
183
184     ret    ; put2digit end
185
186
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
187 ; *-----*
188 ; * Clearing the LCD Display Image *
189 ; *-----*
190 cls:
191     push   OCR           ; Pushes the OCR value onto the stack
192     mov    #osc_rc,OCR   ; Specifies the system clock
193
194     mov    #0,XBNK      ; Specifies the display RAM bank address (BANK0)
195     call   cls_s        ; Clears the data in that bank
196
197     mov    #1,XBNK      ; Specifies the display RAM bank address (BANK1)
198     call   cls_s        ; Clears the data in that bank
199     pop    OCR          ; Pops the OCR value off of the stack
200
201     ret                ; cls end
202
203 cls_s:                    ; *** Clearing One Bank of Display RAM ***
204     mov    #80h,r2      ; Points the indirect addressing register at the
                        ; start of display RAM
205     mov    #80h,b       ; Sets the number of loops in loop counter b
206 loop3:
207     mov    #0,@r2      ; Writes "0" while incrementing the address
208     inc    r2           ;
209     dbnz  b,loop3      ; Repeats until b is "0"
210
211     ret                ; cls_s end
212
213
214 ; *-----*
215 ; * Displaying One Character in a Specified Position*
216 ; * Inputs: acc: Character code *
217 ; *                c: Horizontal position of character*
218 ; *                b: Vertical position of character*
219 ; *-----*
220 patch:
221     push   XBNK
222     push   acc
223     call   locate      ; Calculates display RAM address according to coordinates
224     pop    acc
225     call   put_chara   ; Displays one character
226     pop    XBNK
227
228     ret                ; patch end
229
230
```



```
231 locate: ; **** Calculating the Display RAM Address According to the Display
          Position Specification ****
232 ; ** Inputs: c: Horizontal position (0 to 5) b: Vertical position (0 to 3)
233 ; ** Outputs: r2: RAM address XBANK: Display RAM bank
234
235                ; *** Determining the Display RAM Bank Address ***
236     ld     b                ; Jump to next1 when b >= 2
237     sub    #2                ;
238     bn    PSW,7,next1 ;
239
240     mov    #00h,XBANK        ; Specifies the display RAM bank address (BANK0)
241     br    next2
242 next1:
243     st     b
244     mov    #01h,XBANK        ; Specifies the display RAM bank address (BANK1)
245 next2:
246
247 ; *** Calculating the RAM Address for a Specified Position on the Display ***
248     ld     b                ; b * 40h + c + 80h
249     rol                    ;
250     rol                    ;
251     rol                    ;
252     rol                    ;
253     rol                    ;
254     rol                    ;
255     add    c                ;
256     add    #80h            ;
257     st     r2                ; Stores the RAM address in r2
258
259     ret                    ; locate end
260
261
262 put_chara:
263     push   PSW                ; Pushes the PSW value onto the stack
264     set1   PSW,1            ; Selects data RAM bank 1
265
266                ; *** Calculating the Character Data Address ***
267     rol                    ; (TRH,TRL) = acc*8 + fontdata
268     rol                    ;
269     rol                    ;
270     add    #low(fontdata)    ;
271     st     TRL                ;
272     mov    #0,acc            ;
273     addc   #high(fontdata)    ;
274     st     TRH                ;
275
276     push   OCR                ; Pushes the OCR value onto the stack
277     mov    #osc_rc,OCR        ; Specifies the system clock
278
279     mov    #0,b                ; Offset value for loading the character data
280     mov    #4,c                ; Loop counter
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
281  loop1:
282      ld    b           ; Loads the display data for the first line
283      ldc           ;
284      inc    b           ; Increments the load data offset by 1
285      st    @r2         ; Transfers the display data to display RAM
286      ld    r2          ; Adds 6 to the display RAM address
287      add   #6          ;
288      st    r2          ;
289
290      ld    b           ; Loads the display data for the second line
291      ldc           ;
292      inc    b           ; Increments the load data offset by 1
293      st    @r2         ; Transfers the display data to display RAM
294      ld    r2          ; Adds 10 to the display RAM address
295      add   #10         ;
296      st    r2          ;
297
298      dec    c           ; Decrements the loop counter
299      ld    c           ;
300      bnz   loop1       ; Repeats for 8 lines (four times)
301
302      pop   OCR          ; Pops the OCR value off of the stack
303      pop   PSW         ; Pops the PSW value off of the stack
304
305      ret              ; put_chara end
306
307
308 ; *-----*
309 ; * Character Bit Image Data                               *
310 ; *-----*
311 fontdata:
312      db 07ch, 0e6h, 0c6h, 0c6h, 0c6h, 0ceh, 07ch, 000h      ; '0' 00
313      db 018h, 038h, 018h, 018h, 018h, 018h, 03ch, 000h    ; '1' 01
314      db 07ch, 0c6h, 0c6h, 00ch, 038h, 060h, 0feh, 000h    ; '2' 02
315      db 07ch, 0e6h, 006h, 01ch, 006h, 0e6h, 07ch, 000h    ; '3' 03
316      db 00ch, 01ch, 03ch, 06ch, 0cch, 0feh, 00ch, 000h    ; '4' 04
317      db 0feh, 0c0h, 0fch, 006h, 006h, 0c6h, 07ch, 000h    ; '5' 05
318      db 01ch, 030h, 060h, 0fch, 0c6h, 0c6h, 07ch, 000h    ; '6' 06
319      db 0feh, 0c6h, 004h, 00ch, 018h, 018h, 038h, 000h    ; '7' 07
320      db 07ch, 0c6h, 0c6h, 07ch, 0c6h, 0c6h, 07ch, 000h    ; '8' 08
321      db 07ch, 0c6h, 0c6h, 07eh, 006h, 00ch, 078h, 000h    ; '9' 09
322
323
```

## C. Sample Program Listings

---

```
324 ; *-----*
325 ; * Low Battery Automatic Detection Function ON*
326 ; *-----*
327 BattChkOn:
328     push    PSW                ; Pushes the PSW value onto the stack
329
330     clr1    PSW,1              ; Selects data RAM bank 0
331     mov     #0,acc              ; Detects low battery (0)
332     st      LowBattChk         ; Low battery automatic detection flag (RAM bank
333     0)
334     pop     PSW                ; Pops the PSW value off of the stack
335     ret                                ; BattChkOn end
336
337
338 ; *-----*
339 ; * Low Battery Automatic Detection Function OFF*
340 ; *-----*
341 BattChkOff:
342     push    PSW                ; Pushes the PSW value onto the stack
343
344     clr1    PSW,1              ; Selects data RAM bank 0
345     mov     #0ffh,acc; Does not detect low battery (0ffh)
346     st      LowBattChk         ; Low battery automatic detection flag (RAM bank
347     0)
348     pop     PSW                ; Pops the PSW value off of the stack
349     ret                                ; BattChkOff end
```

# General-purpose Serial Driver

This is a serial transmission/reception program that uses a general-purpose serial driver with a buffer that uses the port 3 interrupt.

If this program is executed on two Visual Memory units, it can be used to send data back and forth between the units and to display the data on their LCDs.

The main routine checks the reception buffer and, if data is found, it gives the highest priority to displaying the received data on the LCD. If the buffer is empty, the program outputs the data to be sent ("0" to "99"). The data that is to be sent is incremented once every 0.5 seconds in response to the base timer interrupt.

The "SioInit" routine in lines 128 to 161 confirm that Visual Memory is connected, initialize the interface, initialize the buffer (RAM), and enable the SIO interrupts. The "SioGet1" routine in lines 203 to 245 get one byte of data that is waiting in the reception buffer.

The SIO reception handler, which operates when an SIO interrupt is received, is the "int\_SioRx" routine in lines 278 to 317. The received data is stored in the buffer.

---

**Caution:** When performing communications using this sample program on both the receiving side and the sending side, no data overflow occurs, but when transferring data consecutively, a wait for a fixed time period should be inserted after each send.

---

If this sample program is used with the previous "Serial Communications (Receiving Side)" sample program, data overflows will occur and smooth communications will not be possible.

### • GHEAD.ASM

```
001 chip      LC868700
002 world     external
003 ; *-----*
004 ; * External header program Ver 1.00*
005 ; *                               05/20-'98*
006 ; *-----*
007
008 public    fm_wrt_ex_exit, fm_vrf_ex_exit
009 public    fm_prd_ex_exit, timer_ex_exit, _game_start, _game_end
010 other_side_symbol fm_wrt_in, fm_vrf_in
011 other_side_symbol fm_prd_in, timer_in, game_end
012
013 extern    main                               ; Symbol in the user program
014 extern    int_BaseTimer                     ; Symbol in the user program
015 extern    int_SioRx                          ; Symbol in the user program
016
```

```
017 ; *-----*
018 ; * Vector table(?) *
019 ; *-----*
020 cseg
021 org 0000h
022 _game_start:
023 ;reset:
024         jmpf  main          ; main program jump
025 org 0003h
026 ;int_03:
027         jmp   int_03
028 org 000bh
029 ;int_0b:
030         jmp   int_0b
031 org 0013h
032 ;int_13:
033         jmp   int_13
034 org 001bh
035 ;int_1b:
036         jmp   int_1b
037 org 0023h
038 ;int_23:
039         jmp   int_23
040 org 002bh
041 ;int_2b:
042         jmp   int_2b
043 org 0033h
044 ;int_33:
045         jmp   int_33
046 org 003bh
047 ;int_3b:
048         jmp   int_3b
049 org 0043h
050 ;int_43:
051         jmp   int_43
052 org 004bh
053 ;int_4b:
054         jmp   int_4b
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
055 ; *-----*
056 ; * interrupt programs      *
057 ; *-----*
058 int_03:
059         reti
060 int_0b:
061         reti
062 int_13:
063         reti
064 int_23:
065         reti
066 int_2b:
067         reti
068 int_33:
069         reti
070 int_3b:
071         jmp         int_SioRx      ; SIO reception interrupt handler
072
073 ; *-----*
074 int_43:
075         reti
076 int_4b:
077         clr1    p3int,1      ; interrupt flag clear
078         reti
079
080 org 0100h
081 ; *-----*
082 ; * flash memory write external program*
083 ; *-----*
084 fm_wrt_ex:
085         change fm_wrt_in
086         fm_wrt_ex_exit:
087         ret
088 org 0110h
089 ; *-----*
090 ; * flash memory verify external program*
091 ; *-----*
092 fm_vrf_ex:
093         change fm_vrf_in
094         fm_vrf_ex_exit:
095         ret
096
097 org 0120h
```

```
098 ; *-----*
099 ; * flash memory page read external program*
100 ; *-----*
101 fm_prd_ex:
102         change fm_prd_in
103         fm_prd_ex_exit:
104             ret
105
106 org 0130h
107 ; *-----*
108 ; * flash memory => timer call external program *
109 ; *-----*
110 int_1b:
111 timer_ex:
112         push    ie
113         clr1    ie,7           ; interrupt prohibition
114         change timer_in
115         timer_ex_exit:
116             call int_BaseTimer ; (User interrupt processing)
117             pop     ie
118             reti
119
120 org 01f0h
121 _game_end:
122         change game_end
123 end
```

**• TIMER1.ASM**

```
001 ; Tab width = 4
002
003 ;-----
004 ; ** Serial Communications Sample 3 (Interrupt-Driven Serial Driver with
005 ; Reception Buffer) **
006 ;
007 ; ·Demonstrates the usage of a serial communications driver with a 16-byte
008 ; reception buffer
009 ; ·Displays the received data values
010 ; ·Sends simple data on a regular cycle
011 ;-----
012 ; 1.01 990208 SEGA Enterprises,LTD.
013 ;-----
013 chip    LC868700           ; Specifies the chip type for the assembler
014 world   external          ; External memory program
015
016 public  main               ; Symbol referenced from ghead.asm
017 public  int_BaseTimer     ; Symbol referenced from ghead.asm
018 public  int_SioRx         ; Symbol referenced from ghead.asm
019
020 extern  _game_end         ; Application end
021
022
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
023 ; **** Definition of System Constants ****
024
025 ; OCR (Oscillation Control Register) settings
026 osc_rc equ 04dh ; Specifies internal RC oscillation for the system clock
027 osc_xt equ 0efh ; Specifies crystal oscillation for the system clock
028
029 LowBattChk equ 06eh ; Low battery detection flag (RAM bank 0)
030
031 SioRxCueSize equ 16 ; Serial communications buffer size
032
033
034 ; *** Data Segment ****
035
036 dseg ; Data segment start
037
038 r0:ds 1 ; Indirect addressing register r0
039 r1:ds 1 ; Indirect addressing register r1
040 r2:ds 1 ; Indirect addressing register r2
041 r3:ds 1 ; Indirect addressing register r3
042 ds 12 ; Other registers
043
044 ; ** For Serial Driver **
045 SioRxCueBehind: ds 1 ; Amount of received data waiting
046 SioRxCueRPnt: ds 1 ; Reception buffer reading point
047 SioRxCueWPnt: ds 1 ; Reception buffer writing point
048 SioRxCue: ds SioRxCueSize ; Reception buffer
049 SioOverRun: ds 1 ; Reception overrun flag
050
051 ; ** Work Areas for Usage Sample **
052 bcount: ds 1 ; Base clock counter
053 work1: ds 1 ; Work 1
054 work2: ds 1 ; Work 2
055
056 work0: ds 1 ; Work (put2digit)
057
058
059 ; *** Code Segment ****
060
061 cseg ; Code segment start
062
```



```
063 ; *-----*
064 ; * Serial Communications Driver Usage Sample *
065 ; * Sends simple data at a regular interval *
066 ; * Displays the received data values on the LCD*
067 ; *-----*
068 main:
069     mov     #0,bcount
070     mov     #0,work1      ; Initial value of transmission data
071     clr1    P3INT,0      ; Masks P3 interrupts
072     call    cls          ; Clears the LCD
073     call    BattChkOff   ; Turns off the low battery automatic detection function
074     call    SioInit      ; Serial communications initialization
075     bnz     finish       ; Ends if VM is not connected
076
077 stlp1:
078                                     ; *** Displaying If Data Has Been Received ***
079     call    SioGet1      ; 1-byte reception
080     be     #0ffh,stnx1   ; Skip if no data has been received
081     bz     stnx3         ; If normal received data is found, go to stnx3
082 error: br     finish     ; Forcibly terminate if an error is detected
083 stnx3:
084     ld     b            ; Load received data from b -> acc
085     mov     #0,c         ; Display coordinate (horizontal)
086     mov     #0,b         ; Display coordinate (vertical)
087     call    put2digit    ; Displays numeric value on the LCD
088     br     stlp1        ; Continues to repeat as long as there is received data
089 stnx1:
090
091     set1    pcon,0       ; Waits until next interrupt
092
093                                     ; *** Sending Simple Data at a Regular Interval ***
094     ld     bcount       ; Base timer counter value
095     be     work2,stnx4   ; Does not send if unchanged
096     st     work2        ; Updates work2
097
098     ld     work1        ; Loads the transmission data
099     call    SioPut1     ; Sends
100
101     inc     work1        ; Updates the transmission data
102     ld     work1        ; (Sends values form 0 to 99, in sequence)
103     bne    #100,stnx2   ;
104     mov     #0,work1    ;
105 stnx2:
106 stnx4:
107
108                                     ; ** [M] (mode) Button Check **
109     ld     P3
110     bn     acc,6,finish  ; If the [M] button is pressed, the application ends
111
112     jmp     stlp1       ; Repeat
113
114 finish:
115                                     ; ** Application End Processing **
116     call    SioEnd      ; Serial communications end processing
117     call    BattChkOn   ; Turns on the low battery automatic detection function
118     jmp     _game_end   ; Application end
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
119 ; *=====*
120 ; ***** Simple Serial Communications Driver *****
121 ; *=====*
122
123 ; *-----*
124 ; * Serial communications initialization *
125 ; *
126 ; * This sample assumes that the system clock is in crystal mode.*
127 ; *-----*
128 SioInit:
129                                     ; **** VM Connection Check ****
130     ld     P7                        ; Checks the connection status
131     and    %#00001101                ; Checks P70, P72, P73
132     be     %#00001000,next3          ; P70 = 0, P72 = 0, P73 = 1
133                                     ; To next3 if connected
134     mov    #0ffh,acc                 ; If not connected, abnormal end with acc = 0ffh
135     ret                                     ; SioInit end
136 next3:
137
138                                     ; **** Serial Communications Initialization ****
139     mov    #0,SCON0                  ; Specifies output as 'LSB first'
140     mov    #0,SCON1                  ; Specifies input as 'LSB first'
141     mov    #0ddh,SBR                 ; Sets the transfer rate
142     clr1  P1,0                       ; Clears the P10 latch (P10/S00)
143     clr1  P1,2                       ; Clears the P12 latch (P12/SCK0)
144     clr1  P1,3                       ; Clears the P13 latch (P13/S01)
145
146     mov    %#00000101,P1FCR          ; Sets the pin functions
147     mov    %#00000101,P1DDR          ; Sets the pin functions
148
149     mov    #0,SBUF0                  ; Clears the transfer buffer
150     mov    #0,SBUF1                  ; Clears the transfer buffer
151
152     mov    #0,acc
153     st     SioRxCueBehind             ; Resets amount of received data waiting
154     st     SioRxCueRPnt              ; Reception buffer reading point
155     st     SioRxCueWPnt              ; Reception buffer writing point
156     st     SioOverRun                ; Resets reception overrun flag
157
158     set1  SCON1,0                    ; Receiving side transfer end interrupt enable
159     set1  SCON1,3                    ; Receiving standby
160
161     ret                                ; SioInit end
162
163
```

## C. Sample Program Listings

```
164 ; *-----*
165 ; * Serial Communications End *
166 ; *-----*
167 SioEnd: ; **** Serial Communications End Processing ****
168
169     mov     #0,SCON0      ; SCON0 = 0
170     mov     #0, SCON1    ; SCON1 = 0
171     mov     #0bfh,P1FCR  ; P1FCR = 0bfh
172     mov     #0a4h,P1DDR  ; P1DDR = 0a4h
173
174     ret     ; SioEnd end
175
176
177 ; *-----*
178 ; * Sending 1 Byte *
179 ; * *
180 ; * Inputs: acc: Transmission data *
181 ; *-----*
182 SioPut1:
183     push   acc           ; Pushes the transmission data onto the stack
184 splp1: ld     SCON0      ; Waits until any previous transfer is completed
185     bp     acc,3,splp1   ;
186     pop    acc           ; Pops the transmission data off of the stack
187
188     st     SBUF0        ; Sets the data to be transferred
189     set1   SCON0,3      ; Starts sending
190
191     ret     ; SioPut1 end
192
193
194 ; *-----*
195 ; * Reading 1 Byte from the Reception Buffer (Asynchronous Reception)*
196 ; * *
197 ; * Outputs: acc: 0 = Normal end *
198 ; * * Offh = No received data *
199 ; * * Ofeh = Buffer overflow *
200 ; * * Ofdh = Overrun error *
201 ; * * b: Received data (Valid only in the case of normal end.)*
202 ; *-----*
203 SioGet1:
204     ; ** Waiting Data Amount check**
205     ld     SioRxCueBehind ; Waiting amount of data
206     bnz   sgnx1          ; When waiting amount != 0
207     mov   #0ffh,acc      ; When waiting amount == 0
208     ret   ; Returns when acc = 0ffh (no received data)
209 sgnx1:
210     ; ** Buffer Overflow Detection **
211     ; SioRxCueBehind - SioRxCueSize
212     be    #SioRxCueSize,sgnx3 ; SioRxCueBehind == SioRxCueSize
213     bp    PSW,7,sgnx3     ; SioRxCueBehind < SioRxCueSize
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
214                                     ; SioRxCueBehind > SioRxCueSize
215     mov     #0feh,acc                ; When the buffer capacity has been exceeded
216     ret                                     ; Return when acc = 0feh (buffer overflow)
217 sgnx3:
218                                     ; ** Overrun Error Detection **
219     ld      SioOverRun                ; Overrun flag
220     bz      sgnx4                    ; Not detected
221     mov     #0fdh,acc                ; Detected
222     ret                                     ; Return when acc = 0fdh (overrun error)
223 sgnx4:
224
225     dec     SioRxCueBehind            ; dec waiting amount
226
227                                     ; ** Calculating the received data reading point
228     ld      SioRxCueRPnt             ; r0 = SioRxCue + SioRxCueRPnt
229     add     #SioRxCue                ;
230     st      r0                       ;
231
232     inc     SioRxCueRPnt             ; inc data reading point
233
234                                     ; ** If reading point = buffer size,
235                                     ; ** then reading point is reset to 0
236     ld      SioRxCueRPnt
237     bne     #SioRxCueSize,sgnx2      ; When SioRxCueRPnt != SioRxCueSize
238     mov     #0,SioRxCueRPnt         ; When SioRxCueRPnt == SioRxCueSize
239 sgnx2:
240
241     ld      @r0                      ; Loads the input data into acc
242     st      b                        ; Stores the value in b
243     mov     #0,acc                  ; acc = acc = 0 (normal end, data exists)
244
245     ret                                     ; SioGet1 end
246
247
248 ; *-----*
249 ; * Reading 1 Byte from the Reception Buffer      *
250 ; * (If there is no received data, this routine waits until data is received)*
251 ; *
252 ; * Outputs: acc: 0 = Normal end                  *
253 ; *                0feh = Buffer overflow          *
254 ; *                0fdh = Overrun error          *
255 ; *                b: Received data (Valid only in the case of normal end.)*
256 ; *-----*
257 SioGet1W:
258     call   SioGet1                    ; Asynchronous reception
259     be     #0ffh,SioGet1W            ; Waits until data is received
260
261     ret                                     ; SioGet1W end
262
263
```

```
264 ; *-----*
265 ; * Getting the Amount of Data Waiting in the Reception Buffer*
266 ; *
267 ; * Output: acc: Amount of data (bytes)
268 ; *-----*
269 SioGetRxLen:
270     ld     SioRxCueBehind    ; Amount waiting
271
272     ret
273
274
275 ; *-----*
276 ; * SIO Reception Interrupt Handler
277 ; *-----*
278 int_SioRx:
279     push  acc                ; Pushes the register to be used onto the stack
280     push  PSW                ;
281     setl  PSW,1              ; Selects data RAM bank 1
282     push  r0                 ; Pushes the register onto the stack
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299 isnx1:
300
301
302
303
304
305
306
307
308
309 isnx2:
310
311
312
313
314
315
316
317
318
319
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
320 ; *-----*
321 ; * Displaying a Two-digit Value *
322 ; * Inputs: acc: Numeric value *
323 ; *           c: Horizontal position of character *
324 ; *           b: Vertical position of character *
325 ; *-----*
326 put2digit:
327     push    b           ; Pushes the coordinate data onto the stack
328     push    c           ;
329     st      c           ; Calculates the tens digit and the ones digit
330     xor     a           ; ( acc = acc/10, work0 = acc mod 10 )
331     mov     #10,b       ;
332     div                    ;
333     ld      b           ;
334     st      work0       ; Stores the ones digit in work0
335     ld      c           ;
336     pop     c           ; Pops the coordinate values into (c, b)
337     pop     b           ;
338     push   b           ; Pushes the coordinates onto the stack again
339     push   c           ;
340     call   putchar     ; Displays the tens digit
341     ld     work0       ; Loads the ones digit
342     pop    c           ; Pops the coordinate values into (c, b)
343     pop    b           ;
344     inc    c           ; Moves the display coordinates to the right
345     call   putchar     ; Displays the ones digit
346
347     ret                    ; put2digit end
348
349
350 ; *-----*
351 ; * Clearing the LCD Display Image *
352 ; *-----*
353 cls:
354     push   OCR          ; Pushes the OCR value onto the stack
355     mov    #osc_rc,OCR  ; Specifies the system clock
356
357     mov    #0,XBNK     ; Specifies the display RAM bank address (BANK0)
358     call   cls_s       ; Clears the data in that bank
359
360     mov    #1,XBNK     ; Specifies the display RAM bank address (BANK1)
361     call   cls_s       ; Clears the data in that bank
362     pop    OCR         ; Pops the OCR value off of the stack
363
364     ret                    ; cls end
365
366 cls_s:                    ; *** Clearing One Bank of Display RAM ***
367     mov    #80h,r2     ; Points the indirect addressing register at the start
                          ; of display RAM
368     mov    #80h,b      ; Sets the number of loops in loop counter b
```

```
369 loop3:
370     mov    #0,@r2      ; Writes "0" while incrementing the address
371     inc    r2          ;
372     dbnz  b,loop3     ; Repeats until b is "0"
373
374     ret                ; cls_s end
375
376
377 ; *-----*
378 ; *   Displaying One Character in a Specified Position*
379 ; *   Inputs: acc: Character code                      *
380 ; *               c: Horizontal position of character*
381 ; *               b: Vertical position of character*
382 ; *-----*
383 patch:
384     push  XBNK
385     push  acc
386     call  locate      ; Calculates display RAM address according
                       ; to coordinates
387     pop   acc
388     call  put_chara   ; Displays one character
389     pop   XBNK
390
391     ret                ; patch end
392
393
394 locate: ; **** Calculating the Display RAM Address According to the Display
          ; Position Specification ****
395 ; ** Inputs: c: Horizontal position (0 to 5) b: Vertical position (0 to 3)
396 ; ** Outputs: r2: RAM address XBNK: Display RAM bank
397
398 ; *** Determining the Display RAM Bank Address
399 ; ***
400     ld    b           ; Jump to next1 when b >= 2
401     sub   #2          ;
402     bn    PSW,7,next1 ;
403
404     mov   #00h,XBNK   ; Specifies the display RAM bank address (BANK0)
405     br   next2
406 next1:
407     st    b           ;
408     mov   #01h,XBNK   ; Specifies the display RAM bank address (BANK1)
409 next2:
410 ; *** Calculating the RAM Address for a Specified Position on the Display ***
411     ld    b           ; b * 40h + c + 80h
412     rol                ;
413     rol                ;
414     rol                ;
415     rol                ;
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
416         rol                ;
417         rol                ;
418         add     c           ;
419         add     #80h       ;
420         st      r2         ; Stores the RAM address in r2
421
422         ret                ; locate end
423
424
425 put_chara:
426         push   PSW         ; Pushes the PSW value onto the stack
427         setl   PSW,1       ; Selects data RAM bank 1
428
429                 ; *** Calculating the Character Data Address ***
430         rol                ; (TRH,TRL) = acc*8 + fontdata
431         rol                ;
432         rol                ;
433         add     #low(fontdata);
434         st      TRL        ;
435         mov     #0,acc      ;
436         addc   #high(fontdata);
437         st      TRH        ;
438
439         push   OCR         ; Pushes the OCR value onto the stack
440         mov     #osc_rc,OCR ; Specifies the system clock
441
442         mov     #0,b        ; Offset value for loading the character data
443         mov     #4,c        ; Loop counter
444 loop1:
445         ld      b          ; Loads the display data for the first line
446         ldc                ;
447         inc     b          ; Increments the load data offset by 1
448         st     @r2        ; Transfers the display data to display RAM
449         ld     r2         ; Adds 6 to the display RAM address
450         add     #6         ;
451         st     r2         ;
452
453         ld     b          ; Loads the display data for the second line
454         ldc                ;
455         inc     b          ; Increments the load data offset by 1
456         st     @r2        ; Transfers the display data to display RAM
457         ld     r2         ; Adds 10 to the display RAM address
458         add     #10        ;
459         st     r2         ;
460
461         dec     c          ; Decrements the loop counter
462         ld     c          ;
463         bnz    loop1      ; Repeats for 8 lines (four times)
464
465         pop     OCR        ; Pops the OCR value off of the stack
466         pop     PSW        ; Pops the PSW value off of the stack
467
468         ret                ; put_chara end
469
470
```



## C. Sample Program Listings

```
471 ; *-----*
472 ; * Character Bit Image Data *
473 ; *-----*
474 fontdata:
475     db 07ch, 0e6h, 0c6h, 0c6h, 0c6h, 0ceh, 07ch, 000h      ; 0
476     db 018h, 038h, 018h, 018h, 018h, 018h, 03ch, 000h    ; 1
477     db 07ch, 0c6h, 0c6h, 00ch, 038h, 060h, 0feh, 000h    ; 2
478     db 07ch, 0e6h, 006h, 01ch, 006h, 0e6h, 07ch, 000h    ; 3
479     db 00ch, 01ch, 03ch, 06ch, 0cch, 0feh, 00ch, 000h    ; 4
480     db 0feh, 0c0h, 0fch, 006h, 006h, 0c6h, 07ch, 000h    ; 5
481     db 01ch, 030h, 060h, 0fch, 0c6h, 0c6h, 07ch, 000h    ; 6
482     db 0feh, 0c6h, 004h, 00ch, 018h, 018h, 038h, 000h    ; 7
483     db 07ch, 0c6h, 0c6h, 07ch, 0c6h, 0c6h, 07ch, 000h    ; 8
484     db 07ch, 0c6h, 0c6h, 07eh, 006h, 00ch, 078h, 000h    ; 9
485
486
487 ; *-----*
488 ; * Low Battery Automatic Detection Function ON*
489 ; *-----*
490 BattChkOn:
491     push    PSW                ; Pushes the PSW value onto the stack
492     clr1    PSW,1              ; Selects data RAM bank 0
493
494     mov     #0,LowBattChk      ; Detects low battery (0)
495
496     pop     PSW                ; Pops the PSW value off of the stack
497     ret                          ; BattChkOn end
498
499
500 ; *-----*
501 ; * Low Battery Automatic Detection Function OFF*
502 ; *-----*
503 BattChkOff:
504     push    PSW                ; Pushes the PSW value onto the stack
505     clr1    PSW,1              ; Selects data RAM bank 0
506
507     mov     #0ffh,LowBattChk   ; Does not detect low battery (0ffh)
508
509     pop     PSW                ; Pops the PSW value off of the stack
510     ret                          ; BattChkOff end
511
512
513 ; *-----*
514 ; * Base Timer Interrupt Handler *
515 ; *-----*
516 int_BaseTimer:
517     clr1    bocr,1             ; Clears the base timer interrupt source
518     inc     bcount             ; Counter ++
519     ret                          ; User interrupt processing end
```

# Reading and Writing Flash Memory

This sample program writes, reads, and verifies flash memory, and displays the characters "SEGA" one at a time upon the completion of each phase.

Lines 60 to 78 prepare, in RAM, the data that will be written in flash memory. The data values range from 0 to 128, and are set in addresses 10H through 8FH in bank 1 of RAM, using the indirect address register. Once the data preparation phase is completed, the program displays an "S" on the LCD.

Lines 91 to 102 set the parameters for calling system BIOS, and disable automatic low battery detection.

Lines 104 to 115 switch the system clock to 1/6 RC before calling the system BIOS. The system clock is switched back to the original clock (crystal oscillation) after the system BIOS has been called.

After switching the clock, the program enables automatic low battery detection and then displays an "E".

---

**Caution:** Disable all interrupts, including the base timer, while flash memory is being accessed. Because the built-in clock function is used by the base timer, keep the length of time that interrupts are disabled as short as possible.  
When writing to flash memory, set the system clock to 1/6 RC. When loading from flash memory, 1/12 RC is also permissible.

---

Lines 128 to 146 uses the system BIOS' verify function to compare the data that was written into flash memory with the data in RAM. If the data matches exactly, the program displays a "G" on the LCD. If the data does not match and the system BIOS returned an error, the program does not display a "G" but does execute the next phase.

Lines 159 to 173 load into RAM the data that was written in flash memory. The data that is loaded is verified by the program's own compare routine starting in line 172. If the data matches completely, the program displays an "A" on the LCD and then terminates. If the data does not match, the program terminates without displaying an "A".

If the "G" or "A" is not displayed, it indicates that the data was corrupted by an earlier access t flash memory.

The data in line 357 and beyond is where the data is to be written (flash memory).

---

**Caution:** When writing to flash memory, be certain to provide an area within the application itself where the data can be written. Writing to flash memory outside of the application is prohibited.

---

Because flash memory accesses are always conducted in units of 128 bytes, "ORG" in line 364 aligns the data with a 128-byte boundary.

---

**Caution:** The "DS" command, an assembler pseudo-instruction, cannot be used to allocate an area in flash memory. The "DS" command can only be used for RAM areas.

---

```
001 ; Tab width = 4
002
003 ;-----
004 ; ** Flash Memory Usage Sample 1 **
005 ;
006 ; This sample writes and verifies data in flash memory, and then reads and
    verifies the data.
007 ; If all operations are performed correctly, the characters "SEGA" appear on the LCD.
008 ;-----
009 ; 1.01 990208 SEGA Enterprises,LTD.
010 ;-----
011
012 chip    Lc868700          ; Specifies the chip type for the assembler
013 world   external        ; External memory program
014
015 public  main             ; Symbol referenced from ghead.asm
016
017 extern  _game_end        ; Symbol reference to ghead.asm
018 extern  fm_wrt_ex, fm_vrf_ex, fm_prd_ex ; Symbol reference to ghead.asm
019
020
021 ; **** Definition of System Constants *****
022
023                ; OCR (Oscillation Control Register) settings
024 osc_rc  equ    04dh      ; Specifies internal RC oscillation for the system clock (1/12)
025 osc_rcfwequ  0cdh      ; Specifies internal RC oscillation for the system clock (1/6)
026 osc_xt  equ    0efh      ; Specifies crystal oscillation for the system clock
027
028          LowBattChkequ  06eh          ; Low battery detection flag (RAM bank 0)
029
030          fmflag  equ    07ch      ; Flash memory write end detection method
031          fmbank  equ    07dh      ; Flash memory bank switching
032          fmadd_h  equ    07eh      ; Flash memory upper address
033          fmadd_l  equ    07fh      ; Flash memory lower address
034
035          fmbuff  equ    080h      ; Start of buffer for flash memory reading/writing
036
037 ; *** Data Segment *****
038
039          dseg                ; Data segment start
040
041 r0:ds   1                    ; Indirect addressing register r0
042 r1:ds   1                    ; Indirect addressing register r1
043 r2:ds   1                    ; Indirect addressing register r2
044 r3:ds   1                    ; Indirect addressing register r3
045 ds     12                    ; Other registers reserved for the system
046
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
047 ; *** Code Segment *****
048
049             cseg             ; Code segment start
050
051 ; *-----*
052 ; * User program                               *
053 ; *-----*
054 main:
055     call    cls             ; Clears the LCD display image
056
057
058                                     ; Preparing Data for the Test Write
059                                     ; Prepares 128 bytes of data from 10h to 8fh in fmbuff
060
061     push   PSW             ; Pushes the PSW value onto the stack
062     setl   PSW,1           ; Selects data RAM bank 1
063
064     mov    #fmbuff,r0      ; Moves the read/write buffer address to r0
065     mov    #128,c          ; Loop counter (128 times)
066     mov    #010h,b         ; Initial value of data to be written
067 loop4:
068     ld     b               ; Places the data in the buffer
069     st     @r0             ;
070
071     inc    b               ; Changes the writing test data
072
073     inc    r0              ; Increments the buffer address
074
075     dec    c               ; Decrements the loop counter
076     ld     c
077     bnz   loop4           ; Repeats 128 times
078
079     pop    PSW            ; Pops the PSW value off of the stack
080
081
082                                     ; Displaying "S"
083
084     mov    #1,c           ; Horizontal coordinate
085     mov    #1,b           ; Vertical coordinate
086     mov    #0ah,acc       ; Character code 'S'
087     call   putch          ; Displays a single character
088
089
```

## C. Sample Program Listings

```
090             ; **** Writing to Flash Memory ****
091
092     push    PSW             ; Pushes the PSW value onto the stack
093     set1    PSW,1          ; Selects data RAM bank 1
094
095     mov     #0, fmbank      ; Flash memory bank specification = 0
096     mov     #high(fmarea),fmadd_h ; Writing destination address (upper)
097     mov     #low(fmarea),fmadd_l ; Writing destination address (lower)
098
099     clr1    PSW,1          ; Selects data RAM bank 0
100     mov     #0ffh,acc      ; Does not detect low battery (0ffh)
101     st      LowBattChk     ; Low battery automatic detection flag (RAM bank 0)
102
103     pop     PSW            ; Pops the PSW value off of the stack
104
105     push    OCR             ; Pushes the OCR value onto the stack
106     mov     #osc_rc,OCR    ; Specifies the system clock (RC)
107     call    fm_wrt_ex      ; BIOS "Writing to flash memory"
108     pop     OCR            ; Pops the OCR value off of the stack
109
110     push    PSW             ; Pushes the PSW value onto the stack
111     clr1    PSW,1          ; Selects data RAM bank 0
112     mov     #0,acc         ; Detects low battery (0)
113     st      LowBattChk     ; Low battery automatic detection flag (RAM bank 0)
114     pop     PSW            ; Pops the PSW value off of the stack
115
116
117             ; **** Displaying "E" ****
118
119     mov     #2,c           ; Horizontal coordinate
120     mov     #1,b           ; Vertical coordinate
121     mov     #0bh,acc       ; Character code 'E'
122     call    putch          ; Displays a single character
123
124
125             ; **** Verifying Flash Memory ****
126
127     push    PSW             ; Pushes the PSW value onto the stack
128     set1    PSW,1          ; Selects data RAM bank 1
129
130     mov     #0, fmbank      ; Flash memory bank specification = 0
131     mov     #high(fmarea),fmadd_h ; Address (upper)
132     mov     #low(fmarea),fmadd_l ; Address (lower)
133
134
135     push    OCR             ; Pushes the OCR value onto the stack
136     mov     #osc_rc,OCR    ; Specifies the system clock (RC)
137     call    fm_vrf_ex      ; BIOS "Verifying flash memory"
138     pop     OCR            ; Pops the OCR value off of the stack
139
140     pop     PSW            ; Pops the PSW value off of the stack
141
142     bnz     vrt_bad        ; Branches when write failed
143             ; Displays "G" only when successful
144
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
145                                     ; **** Displaying "G" ****
146
147     mov     #3,c           ; Horizontal coordinate
148     mov     #1,b           ; Vertical coordinate
149     mov     #0ch,acc       ; Character code 'G'
150     call    putch          ; Displays a single character
151 vrt_bad:
152
153
154                                     ; **** Reading Page Data form Flash Memory ****
155
156     push   PSW             ; Pushes the PSW value onto the stack
157     setl   PSW,1           ; Selects data RAM bank 1
158
159     mov     #0,fmbank       ; Flash memory bank specification = 0
160     mov     #high(fmarea),fmadd_h ; Address (upper)
161     mov     #low(fmarea),fmadd_l ; Address (lower)
162
163     push   OCR             ; Pushes the OCR value onto the stack
164     mov     #osc_rc,OCR     ; Specifies the system clock (RC)
165     call    fm_prd_ex      ; BIOS "Reading page data from flash memory"
166     pop    OCR             ; Pops the OCR value off of the stack
167
168     pop    PSW             ; Pops the PSW value off of the stack
169
170
171                                     ; **** Verifying the data that was read ****
172
173     push   PSW             ; Pushes the PSW value onto the stack
174     setl   PSW,1           ; Selects data RAM bank 1
175
176     mov     #fmbuff,r0     ; Moves the read/write buffer address into r0
177     mov     #128,c         ; Loop counter (128 times)
178     mov     #010h,b        ; Initial value for comparison data
179 loop5:
180     ld     b               ; Places the data in the buffer
181     sub    @r0             ; Compares the data
182     bnz    read_bad       ; If a compare error is found, ends without displaying 'A'
183
184     inc    b               ; Changes the data for the write test
185
186     inc    r0              ; Increments the buffer address
187
188     dec    c               ; Decrements the loop counter
189     ld     c
190     bnz    loop5          ; Repeats 128 times
191
192     pop    PSW             ; Pops the PSW value off of the stack
193
194
```

## C. Sample Program Listings

```
195                                     ; **** Displaying "A" ****
196
197     mov     #4,c           ; Horizontal coordinate
198     mov     #1,b           ; Vertical coordinate
199     mov     #0dh,acc       ; Character code 'A'
200     call    putchar       ; Displays a single character
201
202
203 read_bad:
204 loop6:                               ; ** [M] (mode) Button Check **
205     ld      P3
206     bn      acc,6,finish ; If the [M] button is pressed, the application ends
207
208     br      loop6         ; Repeat
209
210     finish:                               ; ** Application End Processing **
211     jmp     _game_end     ; Application end
212
213
214 ; *-----*
215 ; * Clearing the LCD Display Image *
216 ; *-----*
217 cls:
218     push   OCR           ; Pushes the OCR value onto the stack
219     mov    #osc_rc,OCR   ; Specifies the system clock
220
221     mov    #0,XBNK      ; Specifies the display RAM bank address (BANK0)
222     call   cls_s        ; Clears the data in that bank
223
224     mov    #1,XBNK      ; Specifies the display RAM bank address (BANK1)
225     call   cls_s        ; Clears the data in that bank
226     pop    OCR          ; Pops the OCR value off of the stack
227
228     ret                    ; cls end
229
230 cls_s:                               ; *** Clearing One Bank of Display RAM ***
231     mov    #80h,r2      ; Points the indirect addressing register at the start
                          ; of display RAM
232     mov    #80h,b       ; Sets the number of loops in loop counter b
233 loop3:
234     mov    #0,@r2      ; Writes "0" while incrementing the address
235     inc    r2           ;
236     dbnz  b,loop3     ; Repeats until b is "0"
237
238     ret                    ; cls_s end
239
240
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
241 ; *-----*
242 ; * Displaying One Character in a Specified Position*
243 ; * Inputs: acc: Character code *
244 ; *          c: Horizontal position of character*
245 ; *          b: Vertical position of character*
246 ; *-----*
247 patch:
248     push  XBNK
249     push  acc
250     call  locate      ; Calculates display RAM address according to coordinates
251     pop   acc
252     call  put_chara   ; Displays one character
253     pop   XBNK
254
255     ret           ; patch end
256
257
258 locate: ; **** Calculating the Display RAM Address According to the Display Position
          ; Specification ****
259     ; ** Inputs: c: Horizontal position (0 to 5) b: Vertical position (0 to 3)
260     ; ** Outputs: r2: RAM address XBNK: Display RAM bank
261
262     ; *** Determining the Display RAM Bank Address ***
263     ld    b           ; Jump to next1 when b >= 2
264     sub   #2          ;
265     bn   PSW,7,next1 ;
266
267     mov   #00h,XBNK   ; Specifies the display RAM bank address (BANK0)
268     br   next2
269 next1:
270     st    b
271     mov   #01h,XBNK   ; Specifies the display RAM bank address (BANK1)
272 next2:
273
274 ; *** Calculating the RAM Address for a Specified Position on the Display ***
275     ld    b           ; b * 40h + c + 80h
276     rol                   ;
277     rol                   ;
278     rol                   ;
279     rol                   ;
280     rol                   ;
281     rol                   ;
282     add   c           ;
283     add   #80h        ;
284     st    r2          ; Stores the RAM address in r2
285
286     ret           ; locate end
287
288
```



```
289 put_chara:
290     push   PSW           ; Pushes the PSW value onto the stack
291     setl   PSW,1        ; Selects data RAM bank 1
292
293                               ; *** Calculating the Character Data Address ***
294     rol    ; (TRH,TRL) = acc*8 + fontdata
295     rol    ;
296     rol    ;
297     add    #low(fontdata) ;
298     st     TRL           ;
299     mov    #0,acc        ;
300     addc   #high(fontdata) ;
301     st     TRH           ;
302
303     push   OCR           ; Pushes the OCR value onto the stack
304     mov    #osc_rc,OCR   ; Specifies the system clock
305
306     mov    #0,b          ; Offset value for loading the character data
307     mov    #4,c          ; Loop counter
308 loop1:
309     ld     b             ; Loads the display data for the first line
310     ldc    ;
311     inc    b             ; Increments the load data offset by 1
312     st     @r2           ; Transfers the display data to display RAM
313     ld     r2            ; Adds 6 to the display RAM address
314     add    #6           ;
315     st     r2           ;
316
317     ld     b             ; Loads the display data for the second line
318     ldc    ;
319     inc    b             ; Increments the load data offset by 1
320     st     @r2           ; Transfers the display data to display RAM
321     ld     r2            ; Adds 10 to the display RAM address
322     add    #10          ;
323     st     r2           ;
324
325     dec    c             ; Decrements the loop counter
326     ld     c             ;
327     bnz   loop1         ; Repeats for 8 lines (four times)
328
329     pop    OCR           ; Pops the OCR value off of the stack
330     pop    PSW           ; Pops the PSW value off of the stack
331
332     ret    ; put_chara end
333
334
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
335 ; *-----*
336 ; * Character Bit Image Data *
337 ; *-----*
338 fontdata:
339     db 07ch, 0e6h, 0c6h, 0c6h, 0c6h, 0ceh, 07ch, 000h ; '0' 00
340     db 018h, 038h, 018h, 018h, 018h, 018h, 03ch, 000h ; '1' 01
341     db 07ch, 0c6h, 0c6h, 00ch, 038h, 060h, 0feh, 000h ; '2' 02
342     db 07ch, 0e6h, 006h, 01ch, 006h, 0e6h, 07ch, 000h ; '3' 03
343     db 00ch, 01ch, 03ch, 06ch, 0cch, 0feh, 00ch, 000h ; '4' 04
344     db 0feh, 0c0h, 0fch, 006h, 006h, 0c6h, 07ch, 000h ; '5' 05
345     db 01ch, 030h, 060h, 0fch, 0c6h, 0c6h, 07ch, 000h ; '6' 06
346     db 0feh, 0c6h, 004h, 00ch, 018h, 018h, 038h, 000h ; '7' 07
347     db 07ch, 0c6h, 0c6h, 07ch, 0c6h, 0c6h, 07ch, 000h ; '8' 08
348     db 07ch, 0c6h, 0c6h, 07eh, 006h, 00ch, 078h, 000h ; '9' 09
349
350     db 07ch, 0e6h, 076h, 038h, 0dch, 0ceh, 07ch, 000h ; 'S' 0a
351     db 0feh, 0c0h, 0c0h, 0f8h, 0c0h, 0c0h, 0feh, 000h ; 'E' 0b
352     db 07ch, 0e6h, 0c0h, 0dch, 0c6h, 0e6h, 07ch, 000h ; 'G' 0c
353     db 01eh, 036h, 066h, 0c6h, 0c6h, 0feh, 0c6h, 000h ; 'A' 0d
354
355
356 ; *-----*
357 ; * Flash Memory Area for Saving Data *
358 ; *-----*
359     org ((*-1) land 0ff80h) + 80h ; Aligns with 128-byte boundary
360 fmarea:
361     ; Allocates a 128-byte flash memory area
362     db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
363     db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
364     db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
365     db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
366     db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
367     db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
368     db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
369     db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
370
371 end
```

## Low Battery Detection and Saving Data

Visual Memory has a built-in function that automatically detects the low battery condition, displays a message to that effect, and then puts the unit into sleep mode. In this sample program, the application detects the low battery condition on its own, and then saves the data in RAM to flash memory.

The important portion of this program is the low battery detection routine in lines 115 to 125. This routine checks the port 7 low battery flag.

Although the port 7 interrupt could be used, the interrupt processing routine should be designed so that system BIOS is not called.

```
001  ; Tab width = 4
002
003  ;-----
004  ; ** Low Battery Detection and Data Save Sample 1 **
005  ;
006  ; Detects the low battery condition and saves essential data in flash memory
007  ;-----
008  ; 1.01 990208 SEGA Enterprises,LTD.
009  ;-----
010
011  chip      Lc868700          ; Specifies the chip type for the assembler
012  world     external        ; External memory program
013
014  public    main             ; Symbol referenced from ghead.asm
015
016  extern    _game_end        ; Symbol reference to ghead.asm
017  extern    fm_wrt_ex, fm_vrf_ex, fm_prd_ex ; Symbol reference to ghead.asm
018
019
020  ; **** Definition of System Constants ****
021
022                                     ; OCR (Oscillation Control Register) settings
023  osc_rc     equ    04dh      ; Specifies internal RC oscillation for the system clock (1/12)
024  osc_rcfw   equ    0cdh      ; Specifies internal RC oscillation for the system clock (1/6)
025  osc_xt     equ    0efh      ; Specifies crystal oscillation for the system clock
026
027  LowBattChk equ    06eh      ; Low battery detection flag (RAM bank 0)
028
029  fmflag     equ    07ch      ; Flash memory write end detection method
030  fmbank     equ    07dh      ; Flash memory bank switching
031  fmadd_h    equ    07eh      ; Flash memory upper address
032  fmadd_l    equ    07fh      ; Flash memory lower address
033
034  fmbuff     equ    080h      ; Start of buffer for flash memory reading/writing
035
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
036 ; *** Data Segment *****
037
038     dseg                ; Data segment start
039
040     r0:  ds    1        ; Indirect addressing register r0
041     r1:  ds    1        ; Indirect addressing register r1
042     r2:  ds    1        ; Indirect addressing register r2
043     r3:  ds    1        ; Indirect addressing register r3
044         ds    12       ; Other registers reserved for the system
045
046
047 ; *** Code Segment *****
048
049         cseg                ; Code segment start
050
051 ; *-----*
052 ; * User program                                     *
053 ; *-----*
054 main:
055     call  cls                ; Clears the LCD display image
056
057 loop0:                ; Start of test main loop
058
059     ; Application Main Processing
060
061         ; ** [M] (mode) Button Check **
062     ld    P3
063     bn    acc,6,finish ; If the [M] button is pressed, the application ends
064
065         ; ** Battery Status Check **
066     call  ChkBatt          ; Checks the battery status
067     bz    loop0           ; If acc = 0 then battery normal; loops
068
069         ; ** Low Battery Processing **
070     call  prepare          ; Prepares data for test save
071         ; In an actual application, this routine would gather
072         ; the data
073         ; that is to be saved and then place the data
074         ; in the flash ROM write buffer.
075     call  WriteData        ; Writes the data that was prepared in the buffer
076         ; (to be saved)
077         ; to flash memory
078 finish:                ; ** Application End Processing **
079     jmp   _game_end       ; Application end
080
081
```

```
082 ; *-----*
083
084 prepare:                ; **** Preparing Data for Test Save ****
085                        ; Prepares 128 bytes of data from 10h to 8fh in fmbuff
086
087     push   PSW           ; Pushes the PSW value onto the stack
088     setl   PSW,1         ; Selects data RAM bank 1
089
090     mov    #fmbuff,r0    ; Moves the read/write buffer address to r0
091     mov    #128,c        ; Loop counter (128 times)
092     mov    #010h,b       ; Initial value of data to be written
093 loop4:
094     ld     b             ; Places the data in the buffer
095     st     @r0           ;
096
097     inc    b             ; Changes the writing test data
098
099     inc    r0            ; Increments the buffer address
100
101     dec    c             ; Decrements the loop counter
102     ld     c             ;
103     bnz   loop4         ; Repeats 128 times
104
105     pop    PSW          ; Pops the PSW value off of the stack
106
107     ret                    ; prepare end
108
109
110 ; *-----*
111 ; * Detecting Low Battery Status *
112 ; * Outputs:  acc = 0 : Battery status normal *
113 ; *                acc = 0ffh: Low battery *
114 ; *-----*
115 ChkBatt:
116     ld     P7            ; Checks the status of P71
117     bn     acc,1,next3   ; Branches if there is no battery
118
119                        ; ** Battery Exists **
120     mov    #0,acc        ; acc = 0
121     ret                    ; ChkBatt end. acc = 0 is returned if battery exists
122
123 next3:
124     mov    #0ffh,acc     ; acc = 0ffh
125     ret                    ; ChkBatt end. acc = 0ffh is returned if battery exists
126
127
```

## Visual Memory Unit (VMU) Tutorial Revision

---

```
128 ; *-----*
129 ; * Writing Buffer Data to Flash Memory *
130 ; *-----*
131 WriteData: ; **** Writing to Flash Memory ****
132
133     push   PSW           ; Pushes the PSW value onto the stack
134     setl   PSW,1        ; Selects data RAM bank 1
135
136     mov    #0,fmbank; Flash memory bank specification = 0
137     mov    #high(fmarea),fmadd_h ; Writing destination address (upper)
138     mov    #low(fmarea),fmadd_l ; Writing destination address (lower)
139     mov    #0,fmflag ; Detects end by toggle bit method
140
141     clrl  PSW,1         ; Selects data RAM bank 0
142     mov    #0ffh,acc    ; Does not detect low battery (0ffh)
143     st     LowBattChk   ; Low battery automatic detection flag (RAM bank 0)
144
145     pop    PSW         ; Pops the PSW value off of the stack
146
147     push  OCR          ; Pushes the OCR value onto the stack
148     mov   #osc_rc,OCR  ; Specifies the system clock (RC)
149     call  fm_wrt_ex    ; BIOS "Writing to flash memory"
150     pop   OCR          ; Pops the OCR value off of the stack
151
152     push  PSW          ; Pushes the PSW value onto the stack
153     clrl  PSW,1        ; Selects data RAM bank 0
154     mov   #0,acc       ; Detects low battery (0)
155     st    LowBattChk   ; Low battery automatic detection flag (RAM bank 0)
156     pop   PSW         ; Pops the PSW value off of the stack
157
158     ret                ; WriteData end
159
160
161 ; *-----*
162 ; * Clearing the LCD Display Image *
163 ; *-----*
164 cls:
165     push  OCR          ; Pushes the OCR value onto the stack
166     mov   #osc_rc,OCR  ; Specifies the system clock *
167
168     mov   #0,XBNK     ; Specifies the display RAM bank address (BANK0)
169     call  cls_s        ; Clears the data in that bank
170
171     mov   #1,XBNK     ; Specifies the display RAM bank address (BANK1)
172     call  cls_s        ; Clears the data in that bank
173     pop   OCR          ; Pops the OCR value off of the stack
174
175     ret                ; cls end
176
```

```
177 cls_s:                ; *** Clearing One Bank of Display RAM ***
178     mov    #80h,r2     ; Points the indirect addressing register at the start
of
                        display RAM
179     mov    #80h,b      ; Sets the number of loops in loop counter b
180 loop3:
181     mov    #0,@r2     ; Writes "0" while incrementing the address
182     inc   r2          ;
183     dbnz  b,loop3     ; Repeats until b is "0"
184
185     ret                ; cls_s end
186
187
188 ; *-----*
189 ; * Flash Memory Area for Saving Data                *
190 ; *-----*
191     org ((*-1) land 0ff80h) + 80h; Aligns with 128-byte boundary
192 farea:
193     ; Allocates a 128-byte flash memory area
194     db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
195     db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
196     db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
197     db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
198     db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
199     db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
200     db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
201     db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
202
203 end
```





# ***Dreamcast VMU Specifications***



# Table of Contents

<b>VMU Specifications</b> .....	<b>VMU-1</b>
Overview .....	VMU-1
VMU Overview .....	VMU-1
VMU Configuration .....	VMU-2
VMU Functions .....	VMU-4
Mode Settings .....	VMU-7
File Management .....	VMU-9
Management Area .....	VMU-10
Data Area .....	VMU-10
Reserved Area .....	VMU-10
LCD Display .....	VMU-11
XRAM .....	VMU-11
Screen Mode .....	VMU-11
Icons .....	VMU-12
Screen Configuration .....	VMU-12
LCD Characteristics .....	VMU-12
Miscellaneous .....	VMU-12
Executable File Initiation .....	VMU-13
Downloading an Executable File .....	VMU-13
File Size .....	VMU-13
Subroutine .....	VMU-13
Interrupts .....	VMU-14
RAM .....	VMU-14
Save Processing During Executable File Operations .....	VMU-14
Auto Power Off .....	VMU-14
Communications Function .....	VMU-15
Maple Bus Protocol .....	VMU-15
Synchronous Serial Communications .....	VMU-15
Clock Function .....	VMU-16
Settings .....	VMU-16
Alarm Function .....	VMU-17
SLEEP Function .....	VMU-18
SLEEP Operation .....	VMU-18
Buttons .....	VMU-19
Batteries .....	VMU-20
Battery Life .....	VMU-20
Processing When Battery Power Is Exhausted .....	VMU-20
Battery Replacement .....	VMU-20
Postscript .....	VMU-20



# VMU Specifications

## Overview

This document describes the VMU, a peripheral device for the next-generation game system KATANA (Dreamcast).

### VMU Overview

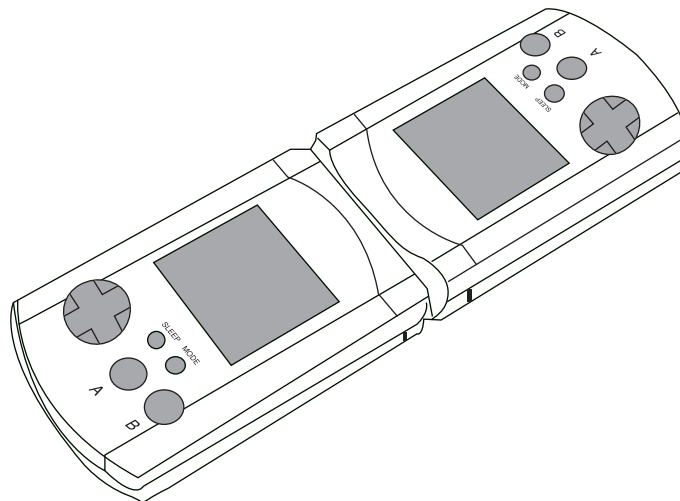
The VMU (Visual Memory Unit) is a memory cartridge that not only stores data, but also includes an LCD display that visually expresses that data.

The VMU connects to KATANA's (preliminary name) special controller, called "SEED" (preliminary name), and can be used to display subscreens during a game and as a memory card that stores game data files.

The VMU can be connected or disconnected while the game machine is on.

When not connected to a controller, the data files stored in the VMU's memory can be displayed and deleted. Files can also be copied from one VMU to another by connecting two VMUs to each other.

Furthermore, by downloading special executable files (programs) from KATANA, the VMU becomes a compact portable game player; two-player games are also possible.



**Figure 1.1** *Conceptual Image of the VMU*

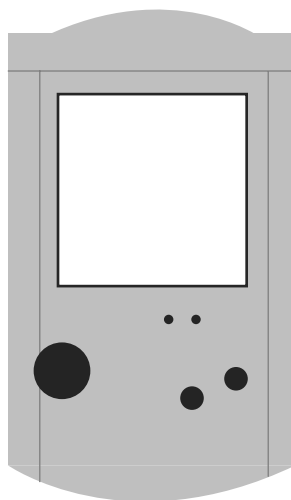
In the top portion of Fig. 1.1, two VMUs are shown connected to each other as they exchange data.

## VMU Specifications

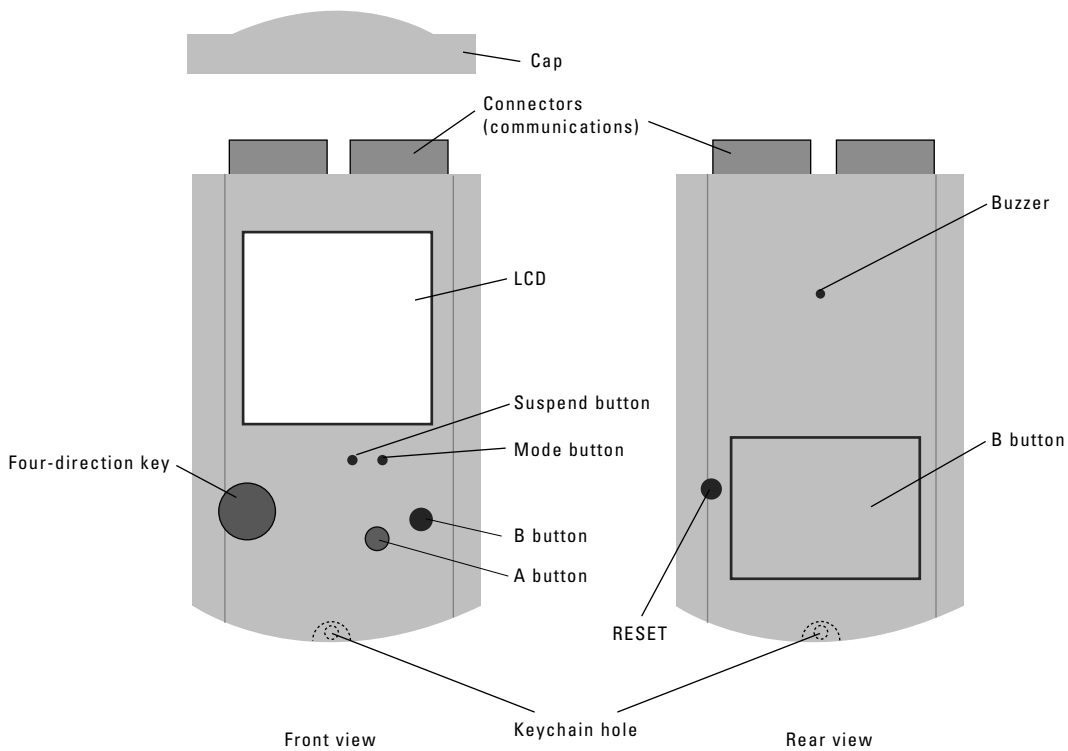
### VMU Configuration

This section describes the VMU configuration.

• Potato Chip (custom IC for the VMU)				
	Core CPU:	8 bits:	Instruction cycle time: When connected to game machine = 1[micro]s When operating on standalone basis = 183[micro]s <b>Note:</b> Operation on a standalone basis is extremely slow in order to minimize battery power consumption.	
	Memory:	Mask-ROM:	16Kbyte	System-BIOS IPL
	:	Flash-EEPROM:	64K	Program code/data area
	:	RAM	64K	Data area (of which 28K are reserved for the system)
	:		512 bytes	General purposes (of which 256 bytes are reserved for the system)
	:	LCD RAM	512bytes	I/O mapping (can also be used as a Maple buffer)
			Bank 1	96 bytes
			Bank 2	96 bytes
			Bank 3	6 bytes (for icons; used by the system)
	Serial I/F:	Uses the following interfaces exclusively:		
		Maple:	LM-Bus	
		Synchronous SIO:	Two 8-bit serial interfaces	
	Timer:	16bit	For Clock	
		16bit(or 8bit x2):	General purpose; of these, 8 bits are used exclusively for pulse generator output for alarms	
	I/O Port:	Input/output:	16 pins (buttons, serial interfaces)	
		Input:	4 pins (control pins)	
	LCD-Driver Controller:	Common:	33 pins	
		Segment:	48 pins	
	• LCD:	LCD:	32 (V) x 48 (H) dots: Monochrome binary	
		Icons:	4 types (File, Game, Time, Attention: used by system)	
	• Buzzer:	Voltage buzzer:	For alarms	
	• Power supply:	Button batteries:	CR2032 x 2	
		External inputs:	+5V +3.3V	
		External outputs:	+3.3V	
	• Buttons:	6 buttons:	Four-direction key, A button, B button, Mode button, Suspend button, SLEEP button	
	• Communications connector:	14 pins:	Serial interface, power supply, control Connected to controller, another VMU, etc.	



**Figure 1.2** External View (preliminary)



**Figure 1.3** External Appearance and configuration (preliminary)

# VMU Specifications

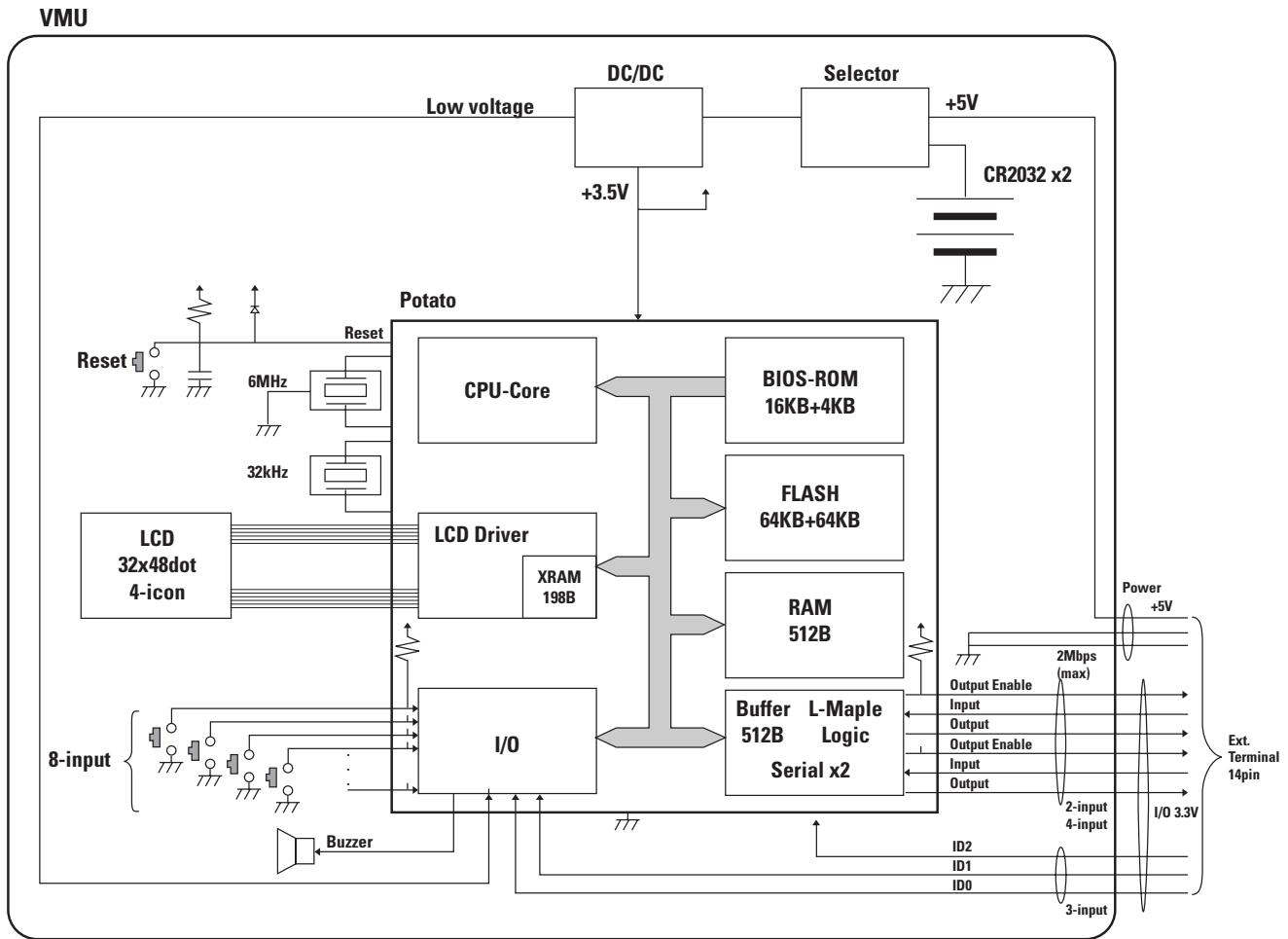


Figure 1.4 Block Diagram (preliminary)

## VMU Functions

When connected to a game machine, the VMU conforms with the Maple Bus 1.0 Standard Specifications, and supports the following function types.

- 1 FT<sub>1</sub> Storage Function
- 2 FT<sub>2</sub> B/W LCD Function
- 3 FT<sub>3</sub> Timer Function

Accordingly, the Function Type (FT) is "00h-00h-00h-0Eh". (FD1 = FT3, FD2 = FT2, FD3 = FT1)



For details, refer to the specifications for each function. An overview of the System-BIOS functions included in the VMU is provided below.

### 1) File management

This function manipulates and manages backup files and program files.

Files are managed in 1-block units (512 bytes), and reads and writes are also performed in block units. FAT operations and file information processing use subroutines in the System-BIOS. For details on file management methods, refer to Chapter , "File Management,".

### 2) LCD display

When the VMU is connected to a game machine, this function only draws graphics (transferring screen image data).

This function conforms with the data format that is stipulated in the Maple Bus Function Type specifications, and sends graphics images from the game machine to the VMU in accordance with the VMU screen configuration, and then BIOS transfers the resulting image to the LCD display RAM (XRAM).

The amount of data required for one screen is 32 dots (V) x 48 dots (H) = 1536 bits = 192 bytes.

When the VMU is operating on a standalone basis, this function handles the drawing of graphics. The icons display the operation mode of the VMU.

File	File management
Game	Executable file initiation
Time	Time display
Attention	Memory access in progress

### 3) Executable file initiation

This function initiates execution of an executable file (program) that was downloaded from a game machine.

This function can only be executed while the VMU is operating on a standalone basis. A program can not be initiated while the VMU is connected to a game machine.

A number of functions that can be provided for executable files are System-BIOS subroutines and can be used by the executable file simply by calling the subroutine.

### 4) Communications

When the VMU is connected to a game machine, communications are handled according to the Maple Bus protocol.

When the VMU is operating on a standalone basis, the VMU supports 8-bit synchronous serial communications for exchanging data with another VMU.

This function is also provided as a subroutine for executable files. (Not finalized)

### 5) Clock

This function uses a timer to measure time.

This function is always operating, whether the VMU is connected to a game machine or is operating on a standalone basis.

### 6) Alarm

This function sounds a buzzer by means of a pulse generator. This function is also provided as a subroutine for executable files. (Not finalized)

This function conforms with the data format that is stipulated in the Maple Bus Function Type specifications, and when the VMU is connected to a game machine, this function allows the game machine to sound the buzzer.

### 7) Mode switching

When the VMU is connected to a game machine, the VMU operation mode can be changed by pressing the mode button.

The mode status is displayed by means of icons.

When the VMU is operating on a standalone basis, the Auto Power Off function can also be used.

### 8) Character font installation

8 dot (V) x 6 dot (H) alphabet, Katakana, and symbol fonts can be installed in the VMU. These fonts cannot be called and displayed from an executable file for the VMU that was downloaded from a game machine.

When the VMU is connected to a game machine and graphics are being displayed from the game machine side, fonts cannot be used.

Instead, transfer the screen image that is to be displayed as is.

Fonts can only be used by the System-BIOS.

# Mode Settings

The operating mode of the VMU is determined by the connection status and the mode button.

**Table 1.1 Modes**

Connection Status	Mode Button (Icon Display)	Operating Mode
<b>Connected to game machine</b>	Off	System mode
	Attention	Flash access in progress
<b>Standalone operation</b>	Game	Executable file initiation
	File	File operations
	Time	Clock display
	Attention	Accessing flash memory

1) System mode

This mode is controlled by the System-BIOS' external control program.

This mode handles communications according to the Maple Bus protocol, memory management, LCD display, and timer management.

2) Game mode

In this mode, the System-BIOS initiates executable files in flash memory.

All processing is controlled by the executable file, except for the Maple Bus protocol.

Transitions from this mode to another mode are also controlled by the executable file.

To execute a mode, transmission, the executable file calls a subroutine from the System-BIOS.

At that point, all of the contents of RAM and the registers are saved to flash memory.

---

**Note:** This save operation requires approximately 8 seconds.

---

3) File mode

This mode is controlled by the System-BIOS' file control program.

This mode can display, copy, and delete files in flash memory through button operations.

Refer to other documents for details on the configuration and operation of the file management screen.

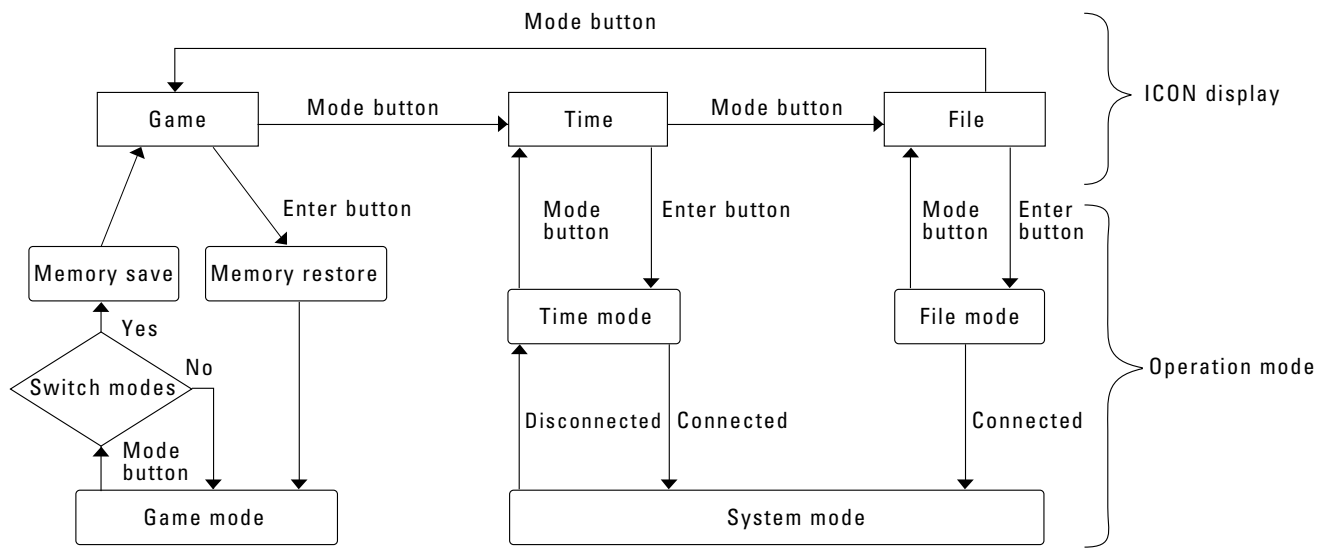
4) Time mode

This mode is controlled by the System-BIOS' timer program. This mode can display a digital clock (showing the hours, minutes, and seconds), and can be used to set the time. When the VMU returns from system mode, it enters this mode.

Transitions among the modes occur in response to changes in the connection status and the Mode button + Enter button being pressed.

However, Game mode can suppress changes in the connection status and the Mode button + Enter button being pressed. The mode cannot be changed while data is being written to the flash memory.

Attention is a warning indicator that lights for Read/Write while flash memory is being accessed.



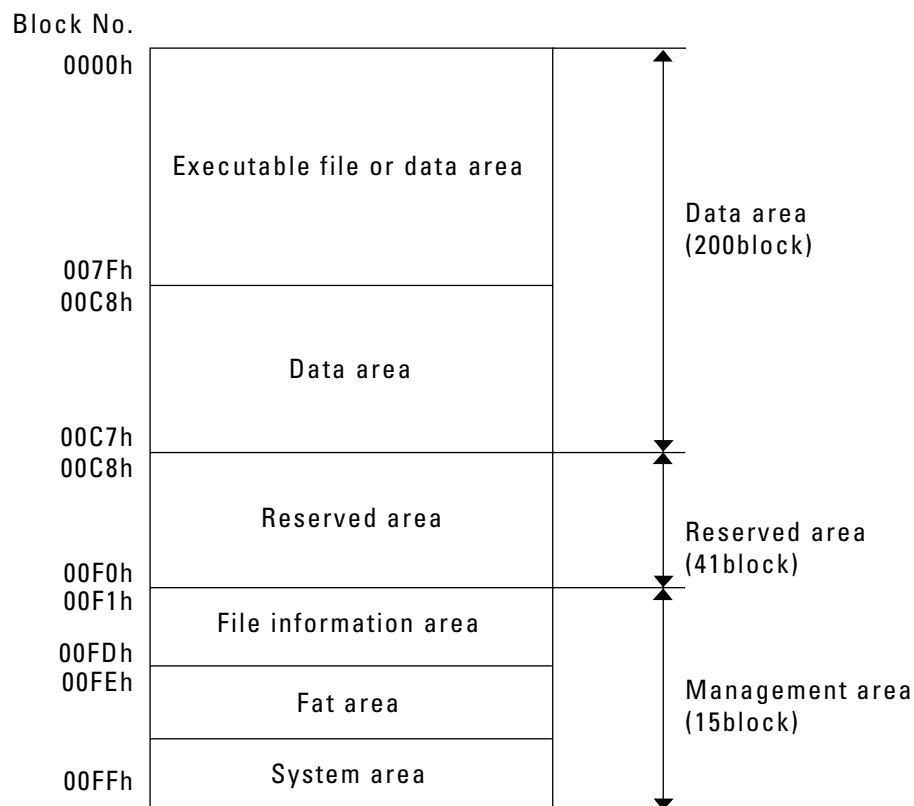
**Figure 1.5** Mode Transitions

## File Management

- File management in the VMU conforms with FT1: Storage Function in the Maple Bus 1.0 Function Type Specifications.
- The size of the VMU flash memory is 128K.
- The minimum read/ write unit for a file is one block (512 bytes); the entire flash memory is divided into 256 blocks.

However, because 56 blocks are used as a system management area, the size of the area that can be used to store data is 200 blocks.

One executable file can exist in one partition, with a maximum size of 0080h blocks (64K: block numbers 0000h to 007Fh).



**Figure 1.6** Memory Map

### **Management Area**

- The 15 blocks at the top of memory (starting from block number 00FFh) are used for the management area.
- The management area is divided into three areas: the system area, the FAT area, and the file information area.
- The system area consists of one block, the FAT area consists of one block, and the file information consists of 13 blocks.
- The system area is write-protected, except during formatting.
- The FAT area has a chain structure in which every two bytes (16 bits) controls one block.
- The file information area allocates 32 bytes to each file, and can therefore manage a maximum of 200 files.
- There is only a root directory; no subdirectories are supported.
- File names consist of 12 bytes (ASCII codes representing up to 12 normal-width characters).

### **Data Area**

- The data area, where data files can be stored, consists of 200 blocks, from block number 0000h to 00C7h.
- Data files are stored starting from 00C7h towards 0000h, while an executable file starts from 0000h.
- The areas from 0000h to 007Fh and from 0080h to 00FFh are controlled through bank switching; switching is performed by the System-BIOS automatically.
- Reading and writing flash memory must always be done by calling the System-BIOS subroutines.

### **Reserved Area**

This area is used by the System-BIOS and in system mode.

## LCD Display

- The LCD display in the VMU conforms with FT[2]: B/W LCD Function in the Maple Bus 1.0 Function Type Specifications.
- The LCD that is built into the VMU consists of a 32-dot (V) × 48-dot (H) dot matrix display, and four icons that indicate the operating mode of the VMU.
- Drawing the LCD is accomplished by storing drawing data in the dedicated drawing RAM.

## XRAM

The LCD's dedicated drawing RAM is called "XRAM."

XRAM consists of three banks; the first and second banks are open to executable files, while the third bank is used by the System-BIOS.

The first bank of XRAM corresponds to the upper half of the LCD (16 × 48 dots), and the second bank of XRAM corresponds to the lower half of the LCD (16 × 48 dots).

One dot on the LCD corresponds to one bit in XRAM. One byte of XRAM corresponds to 8 dots in a horizontal row on the LCD, and 6 bytes consist of one entire horizontal row on the LCD.

## Screen Mode

When the VMU is connected to a game machine, the System-BIOS sends drawing data from the game machine directly to the XRAM as a graphics screen.

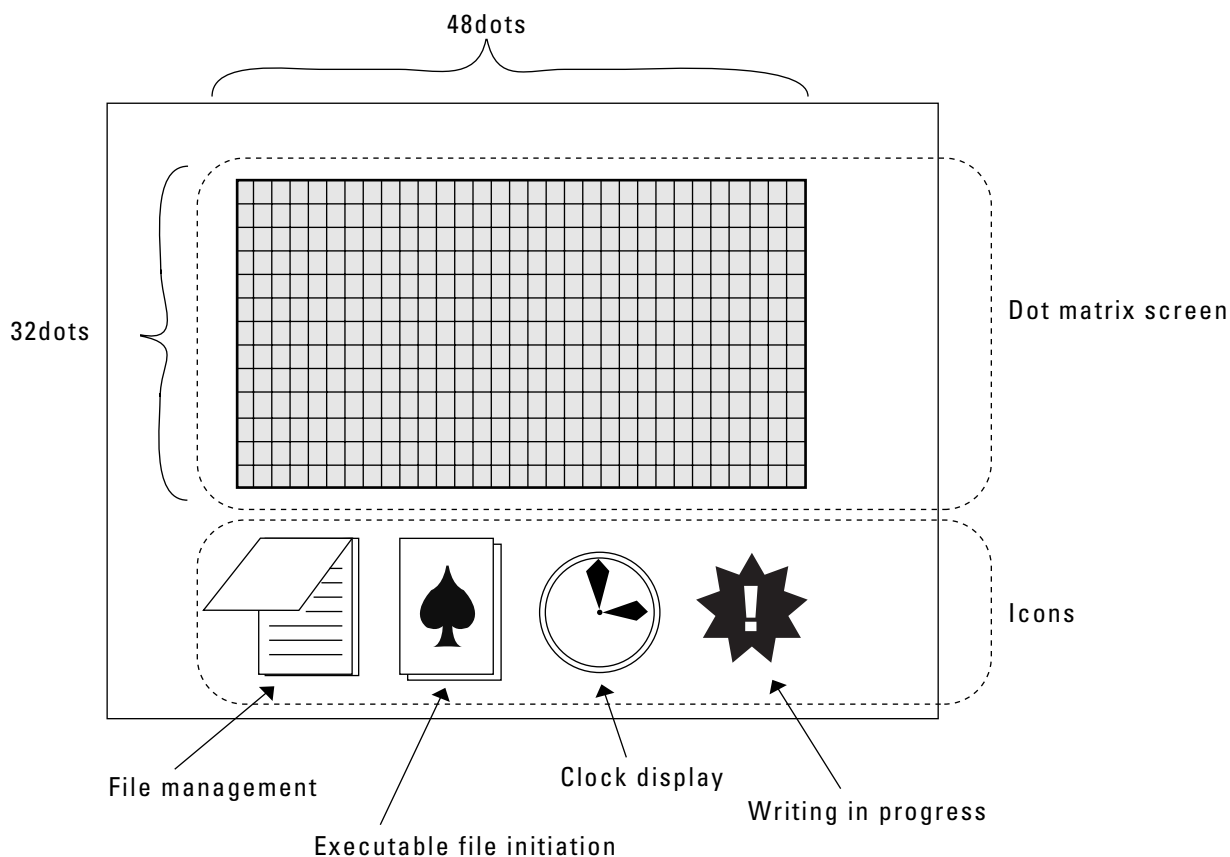
Therefore, when using the VMU's display as a game subscreen, etc., transfer the screen image as is to the VMU.

During standalone operation, the character font in the System-BIOS cannot be used for text display on a graphics screen.

For a graphics screen, write the screen image data as is to XRAM.

### Icons

The System-BIOS uses the icons; use by an executable file is prohibited.



### Screen Configuration

#### LCD Characteristics

The screen refresh concept for the LCD display differs from that for a TV.

Once data is transferred to XRAM, it is displayed on the LCD, but only after a delay due to the response speed of the LCD. When the LCD response is delayed, ghosting or flickering may occur, resulting in a display that is difficult to see. In addition, during standalone operation or when connected to a game machine, differences in the operating speeds result in different LCD display speeds. During standalone operation, the display speed is slower.

The recommended refresh rate for the VMU' LCD is 1Hz for standalone operation and 4Hz when connected to a game machine.

#### Miscellaneous

- There is no contrast adjustment or brightness adjustment for the LCD.
- There is no backlight for the LCD.
- It is not possible to incorporate a design (such as a picture, etc.) in the polarized panel (the back sheet) with a reflective panel that reflects the light in the LCD.



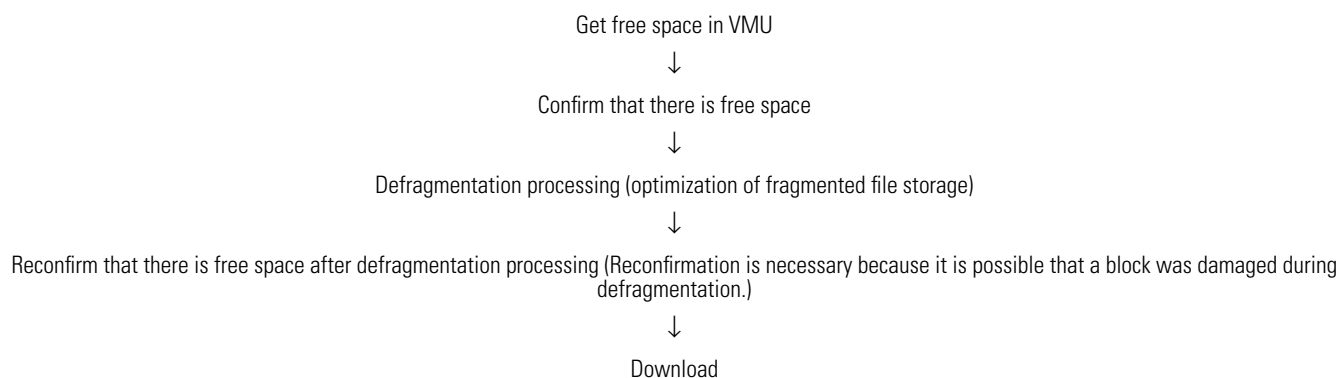
## Executable File Initiation

- This function initiates an executable file that was downloaded from a game machine.
- The VMU can store and initiate only one executable file at a time.
- The System-BIOS includes subroutines that form that VMU functions. Of these subroutines, several are provided for executable files, and an executable file can call these subroutines.
- Program development of an executable file is performed using a VMU emulator (preliminary) that runs under Windows 95.

## Downloading an Executable File

Executable files are stored in flash memory in the area consisting of block numbers 0000h to 007Fh, starting from the 0000h block. When an executable file is downloaded from a game machine application, confirm that there is contiguous free space starting from the 0000h block of the VMU. Even if the free space has been confirmed, it still will not be possible to download an executable file if there is any other file in the area where the executable file is to be stored (the area from block 0000h to the end of the executable file).

Game machine application processing is as described below:



## File Size

- The maximum size of an executable file is 0080h blocks (64K).

## Subroutine

A list of the available subroutines is shown below. (not finalized)

Each subroutine uses a RAM area (in the general-purpose RAM area) as a work area.

- |                        |   |
|------------------------|---|
| 1) Data communications | :Performs synchronized serial communications. |
| 2) Alarm               | :Sounds the buzzer.                           |
| 3) Flash memory write  | :Writes flash memory.                         |
| 4) Flash memory read   | : Reads flash memory.                         |

### Interrupts

A list of external and internal interrupts is provided below. (planned)

Except for the Mode Change interrupt, these interrupts cannot be masked. (planned)

- 1) Low voltage interrupt
- 2) Timer interrupt
- 3) Mode Change interrupt (maskable)
- 4) SLEEP interrupt

### RAM

The RAM areas that executable files can use are shown below.

General-purpose RAM:	000h to 0FFh (bank 1)
I/O mapping RAM:	000h to 1FFh (Set the address to the specified register and read/write one byte at a time.)
XRAM:	Bank 1, bank 2

### Save Processing During Executable File Operations

Data on the midpoint status of an executable file and parameters for an executable file (such as a game) are saved by writing the data to an area within the executable file. When creating an executable file (such as a game), set aside an area within the file for this purpose. Because FAT processing, etc., is not possible due to the hardware design, such data cannot be saved in a separate file.

In order to link the game machine with an application and then use the saved data from an executable file (such as a game), load the executable file from the VMU to the game machine, and then read that portion of the file that contains the saved data.

### Auto Power Off

- The `Auto Power Off` function puts the VMU into the SLEEP state if no buttons are pressed or no communications are received for two minutes.

This function can be enabled/disabled by executable files.

## Communications Function

- The VMU is capable of conducting serial communications with other equipment.
- The VMU supports two serial communications protocols: the Maple Bus protocol and full-duplex synchronous serial communications.
- The System-BIOS switches between the Maple Bus protocol in system mode and synchronous serial communications in standalone operation mode.

### Maple Bus Protocol

- When the VMU is connected to a game machine, the communications connector switches to the Maple Bus protocol side.
- The entire I/O mapping RAM becomes a transmission/receive buffer, and the synchronous serial side stops.
- The physical connection with the game machine is made through an LM-Bus connection, and the VMU becomes an expansion device.
- All processing is performed by the System-BIOS; this function is not accessible from an executable file.
- The transfer speed is 2Mbps.

### Synchronous Serial Communications

- When the VMU is operating on a standalone basis, the communications connector switches to synchronous serial side, and the Maple Bus protocol side stops.
  - There are two synchronous serial interfaces, allowing full duplex communications with other devices.
  - Data is transferred one byte at a time, with a maximum transfer speed of 2.4Kbps. (not finalized)
- This function is available to executable files as a subroutine.

### **Clock Function**

- The clock function in the VMU conforms with FT3: Timer Function in the Maple Bus 1.0 Function Type Specifications.

This function can measure time in 500ms units, using a 32KHz crystal resonator and a dedicated counter.

- The System-BIOS controls the clock function; an executable file can only read the clock function.

### **Settings**

- On the setting screen, set the year, month, day, and time.
- When the VMU is connected to a game machine, the date and time can be set by the game machine through the Maple Bus protocol.

## Alarm Function

- The alarm function in the VMU conforms with FT3: Timer Function in the Maple Bus 1.0 Function Type Specifications.

This function sounds the built-in voltage buzzer.

- Only one alarm can be sounded at one time.

The sound is generated by the pulse generator method; the frequency can be set over a range from 300Hz to 4KHz, and the duty ratio can be set as desired. (planned)

The volume cannot be adjusted. The sound can be turned on and off.

This function is made available for executable programs as a subroutine. (planned)

- When the VMU is connected to a game machine, the alarm function can be set by the game machine through the Maple Bus protocol.

### **SLEEP Function**

In order to reduce power consumption when operating on a standalone basis, the VMU is equipped with a SLEEP function.

The VMU enters the SLEEP state either because the SLEEP button is pressed or because the Auto power Off function was triggered. (Refer to section, "Auto Power Off") To return from the SLEEP state, press the SLEEP button.

### **SLEEP Operation**

When in Timer mode (clock display) or File mode (file management software), the LCD display shuts off and the VMU enters the idle state.

SLEEP processing in Game mode (after an executable file has been initiated is determined by the executable file. (We plan to indicate a recommended processing method.)

The contents of RAM and the registers are retained, except in Time mode. In SLEEP mode, all buttons are disabled except for the SLEEP button.

## Buttons

Four-direction key:	This key is used to move the cursor up, down, left, or right, and to scroll the screen.
A button:	This button is used primarily to finalize selections.
B button:	This button is used primarily to cancel selections.
Mode button:	This button changes the mode during standalone operation. Each time this button is pressed, the mode changes according to the following cycle: File -> Game -> Time -> File -> Game ->...
SLEEP button:	This button changes the mode to the <i>SLEEP</i> state during standalone operation.
Reset button:	This button initiates a “power on” reset, which initializes the entire VMU unit (including the clock, etc.), except for the contents of flash memory.

# Batteries

## Battery Life

The VMU is equipped with two CR2032 batteries for standalone operation.

Battery life depends on the status of executable file operations.

If an executable file is continuously executed, with the LCD display on (refresh rate: 1Hz), no alarm outputs, no use of the communications function, no executable file save processing, and no use of the SLEEP function, the batteries should last for about one week.

The relationship between operational status and battery life is described below. Take battery life into consideration when creating executable files.

Flash memory reads:	This is the normal state of program execution.
LCD display updates:	Battery power consumption increases by a factor of 5 when overwriting XRAM as compared to when reading flash memory. Frequent screen updates have an effect on battery life.
Alarm output:	Consumes an extremely small amount of power.
Flash memory writes:	Consumes 25 times more battery power than when reading flash memory. Saving the operation status and similar processing should be performed as infrequently and in as small amounts as possible.
Data exchanges after an executable file has been initiated:	Such operations consume a tremendous amount of battery power. Simple parameter exchange could be used to reflect the development of game characters, for example.
File exchanges between two VMUs:	Copying an entire file consumes a tremendous amount of battery power. Because the receiving side in particular must write the data in flash memory, a large amount of battery power is consumed. In addition, the larger a file is, the longer the operation will take and the greater that the power consumption will be.

## Processing When Battery Power Is Exhausted

The System-BIOS constantly monitors the battery voltage.

If the batteries are nearing the end of their life while in Game mode (while an executable file is being executed), the System-BIOS saves the contents of RAM and the registers. (planned to be implemented through the library, perhaps)

## Battery Replacement

- The clock settings are initialized when the batteries are replaced.
- Any file that is stored in flash memory is retained.
- When replacing the batteries, always install two brand new CR2032 made by the same manufacturer.
- Make sure that the polarity (+/-) of the batteries is correct when you install them.

## Postscript

The functions of the VMU are subject to change in whole or in part until the release of VMU Specifications Revision 1.0.



***Visual Memory Unit (VMU)***  
***Hardware Manual***



# Table of Contents

<b>Visual Memory Unit Overview .....</b>	<b>VMD-1</b>
VMU Specifications .....	VMD-2
VMU Functions .....	VMD-6
File management .....	VMD-7
Liquid-Crystal Display .....	VMD-7
Starting VMU applications .....	VMD-7
Data transfer .....	VMD-7
Clock .....	VMD-7
Buzzer .....	VMD-8
Operation mode switching .....	VMD-8
Integrated character font .....	VMD-8
Mode Setting .....	VMD-9
System mode .....	VMD-9
Game mode .....	VMD-9
File mode .....	VMD-10
Clock mode .....	VMD-10
File Management .....	VMD-11
Flash memory management area .....	VMD-11
Data area .....	VMD-13
Reserved area .....	VMD-13
LCD Display .....	VMD-14
XRAM .....	VMD-14
Image mode .....	VMD-14
Icon .....	VMD-14
Image configuration .....	VMD-14
LCD characteristics .....	VMD-15
Other important points .....	VMD-15
Starting an Executable File .....	VMD-16
Writing applications for the VMU .....	VMD-16
Transferring an executable file .....	VMD-16
Executable file size .....	VMD-16
OS programs usable by applications .....	VMD-16
RAM .....	VMD-17
Saving application data .....	VMD-17
Auto power-off .....	VMD-18

Communication Functions .....	VMD-19
Maple bus protocol .....	VMD-19
Synchronous serial transfer .....	VMD-19
Clock Function .....	VMD-20
Alarm Function .....	VMD-21
Sleep Function .....	VMD-22
Buttons .....	VMD-23
Batteries .....	VMD-24
Battery life .....	VMD-25
Battery status monitoring .....	VMD-25
Battery replacement .....	VMD-25

## **CPU Features .....** **VMD-27**

Differences to Conventional CPUs .....	VMD-28
Specifications .....	VMD-29
System block diagram .....	VMD-33

## **Internal System Configuration .....** **VMD-35**

Memory Space .....	VMD-35
Program Counter (PC) .....	VMD-36
ROM Space .....	VMD-38
RAM Space .....	VMD-38
Indirect Address Registers .....	VMD-39
Special function registers (SFR) .....	VMD-40
Flash Memory .....	VMD-43
Accumulator .....	VMD-43
B Register, C Register .....	VMD-43
Program Status Word (PSW) .....	VMD-44
Stack Pointer .....	VMD-46
Table Reference Register (TRR) .....	VMD-47
CHANGE Instruction .....	VMD-48
Format .....	VMD-48
Operation .....	VMD-48
Sample program .....	VMD-48

# Peripheral System Configuration . . . . . VMD-49

I/O Ports .....	VMD-49
Port 1 .....	VMD-50
Port 3 .....	VMD-54
Port 7 .....	VMD-56
Timer/Counter 0 (T0) .....	VMD-58
Functions .....	VMD-58
Circuit Configuration .....	VMD-59
Related Registers .....	VMD-60
Circuit Configuration and Operation Principles .....	VMD-69
Timer 1 (T1) .....	VMD-76
Functions .....	VMD-76
Circuit Configuration .....	VMD-77
Related Registers .....	VMD-78
Circuit Configuration and Operation Principles .....	VMD-82
Base Timer .....	VMD-94
Functions .....	VMD-94
Circuit Configuration .....	VMD-95
Related Registers .....	VMD-96
Using the Base Timer .....	VMD-99
Serial Interface .....	VMD-100
Functions and Features .....	VMD-100
Circuit Configuration .....	VMD-102
Related Registers .....	VMD-103
Serial Interface Operation .....	VMD-109
Operation Mode Settings .....	VMD-109
Serial transfer clock .....	VMD-111
Serial Transfer Timing .....	VMD-113
LSB/MSB Switchable Start Sequence .....	VMD-114
Overrun Detection .....	VMD-116
Transfer Bit Length Control .....	VMD-117
Sample Program .....	VMD-117
Dot Matrix LCD Controller .....	VMD-120
Functions .....	VMD-120
Display RAM (XRAM) .....	VMD-120
Display Control Registers .....	VMD-121
External Interrupt Function .....	VMD-128
Circuit Configuration .....	VMD-129
Related Registers .....	VMD-129
Port Interrupt Functions .....	VMD-135
Function .....	VMD-135
Circuit Configuration .....	VMD-135
Related Registers .....	VMD-136
Operation Description .....	VMD-137
State Transition .....	VMD-137
VMU Work RAM .....	VMD-139
Work RAM Control Registers .....	VMD-139
Accessing Work RAM .....	VMD-140
Precautions for Using Work RAM Address Register .....	VMD-140
Flash Memory .....	VMD-142
Features and Functions .....	VMD-142
Accessing Program/Data Area of Flash Memory .....	VMD-142

**Control Functions ..... VMD-143**

Interrupt Functions ..... VMD-143  
    Interrupt Types ..... VMD-144  
    Interrupt Function Operation ..... VMD-145  
    Circuit Configuration ..... VMD-146  
    Related Registers ..... VMD-147  
    Interrupt Priority Ranking ..... VMD-150  
System Clock Generation ..... VMD-151  
    Features and Functions ..... VMD-153  
    Circuit Configuration ..... VMD-154  
    Related Registers ..... VMD-156  
    System Clock Operation Mode ..... VMD-159  
Sleep Function ..... VMD-161  
    Related Registers ..... VMD-162  
    Standby Operation Status ..... VMD-163  
    HALT Mode ..... VMD-164  
Hardware Reset Function ..... VMD-165  
    External Reset Pin Function ..... VMD-166  
    Hardware Status During a Reset ..... VMD-167

**Programs in ROM ..... VMD-171**

System Programs ..... VMD-172  
OS Programs ..... VMD-173  
Headers ..... VMD-174

**Memory Space ..... VMD-175**

**System BIOS Functions ..... VMD-177**

**Subroutine Call Procedure ..... VMD-179**

Processing Contents of Labels ..... VMD-180  
Interaction Between System BIOS and Application ..... VMD-181

**Application Shutdown Procedure When MODE Button is Pressed ..... VMD-183**

Processing Contents of Labels ..... VMD-184  
Interaction Between System BIOS and Application ..... VMD-185

**VMU Initialization ..... VMD-187**

**Subroutine Reference ..... VMD-189**

Flash Memory Access Functions ..... VMD-189  
Subroutine Use Precautions ..... VMD-190  
Flash memory routines ..... VMD-192  
    fm\_prd\_ex(ORG 0120H)  
    Flash memory page data read ..... VMD-192  
    fm\_wrt\_ex(ORG 0100H)  
    Flash memory data write ..... VMD-194  
    fm\_vrf\_ex(ORG 0110H)  
    Flash memory page data verify ..... VMD-195  
Clock Function .....  
VMD-198  
    timer\_ex  
    Clock count-up timer ..... VMD-198

<b>Low Battery Voltage Auto Detection</b> .....	<b>VMD-199</b>
<b>List of Defined Variables</b> .....	<b>VMD-201</b>
<b>Sound Output Method</b> .....	<b>VMD-203</b>
Timer 1 Outline .....	VMD-203
Timer 1 Block Configuration .....	VMD-203
Related Registers .....	VMD-204
Mode Setting .....	VMD-205
8 Bit Counter Mode .....	VMD-206
Output Waveform and Parameter Setting .....	VMD-206
8 Bit Counter Mode Setting .....	VMD-207
Frequency Characteristics .....	VMD-208
Output Frequency Table .....	VMD-208
<b>Sample Program</b> .....	<b>VMD-211</b>
<b>Variable Bit Length Pulse Generator</b> .....	<b>VMD-213</b>
<b>Symbol Table</b> .....	<b>VMD-217</b>
<b>VMU Mode Selection</b> .....	<b>VMD-221</b>
<b>Calculation of Battery Life</b> .....	<b>VMD-223</b>
Methods for Enhancing Battery Life .....	VMD-223
Oscillator Circuit and Current Consumption .....	VMD-224
Oscillation Control Register .....	VMD-224
System Clock Division Ratio Setting .....	VMD-224
Oscillator Circuit Selection .....	VMD-224
Oscillator Circuit Start/Stop .....	VMD-225
Calculating Battery Life .....	VMD-225
Calculating Continuous Operating Time .....	VMD-225
Calculating Battery Life in Days .....	VMD-226
<b>Serial Communication Precautions</b> .....	<b>VMD-229</b>
Serial Communication Timing Chart .....	VMD-229
Measures to Ensure Problem-Free Serial Transfer .....	VMD-230
Mask All Interrupts .....	VMD-230
Set Maximum Send Wait Time .....	VMD-231





# ***Visual Memory Unit Overview***

---

The Visual Memory Unit is a memory cartridge that serves not only for storing data but also for visually displaying information on an integrated LCD. It is connected to the Dreamcast controller (hereafter referred to simply as the “controller”) and is used as a memory card that can store game data and display secondary screens during a game. It can be connected and removed also while the Dreamcast is turned ON.

In the standalone condition (while not connected to the controller), it is possible to display a directory of data files and to perform housekeeping (deleting files). Two VMU units can be connected for operations such as copying files.

By downloading an application from the Dreamcast to the VMU, the VMU can be used as a miniature game machine. Connecting two units for two-player battle-type games is also possible.



**Figure 2.1** *VMU allows two-player battle-type games*

# VMU Specifications

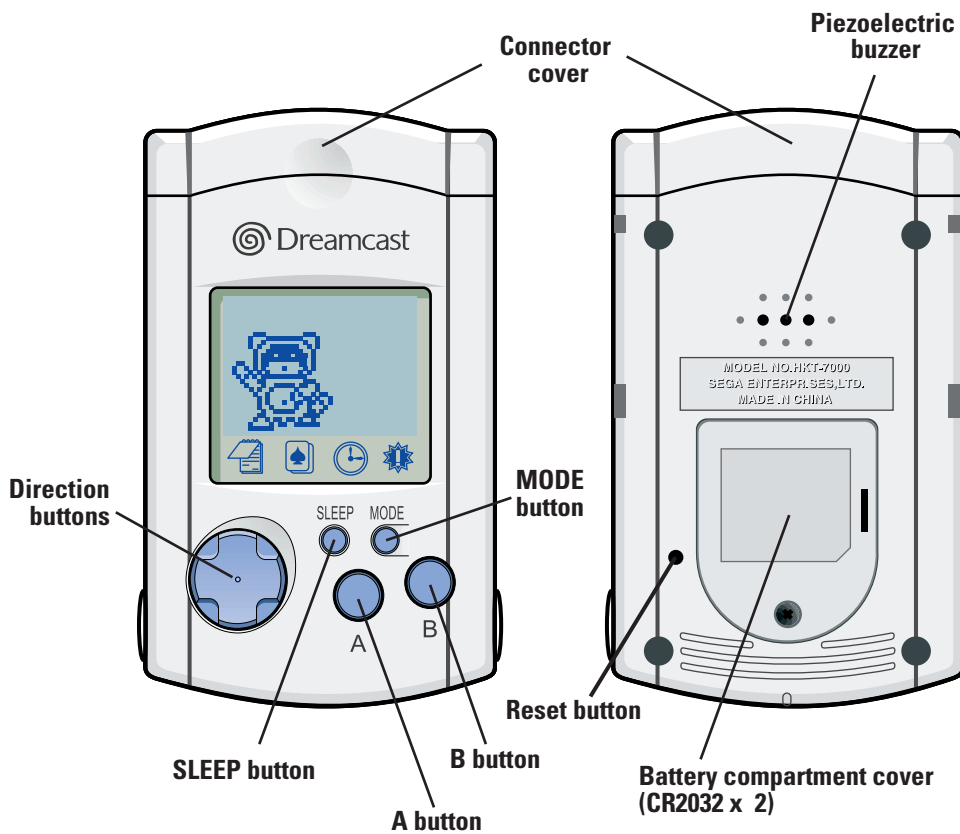
The VMU hardware configuration is shown below.

**Table 2.1 VMU Specifications**

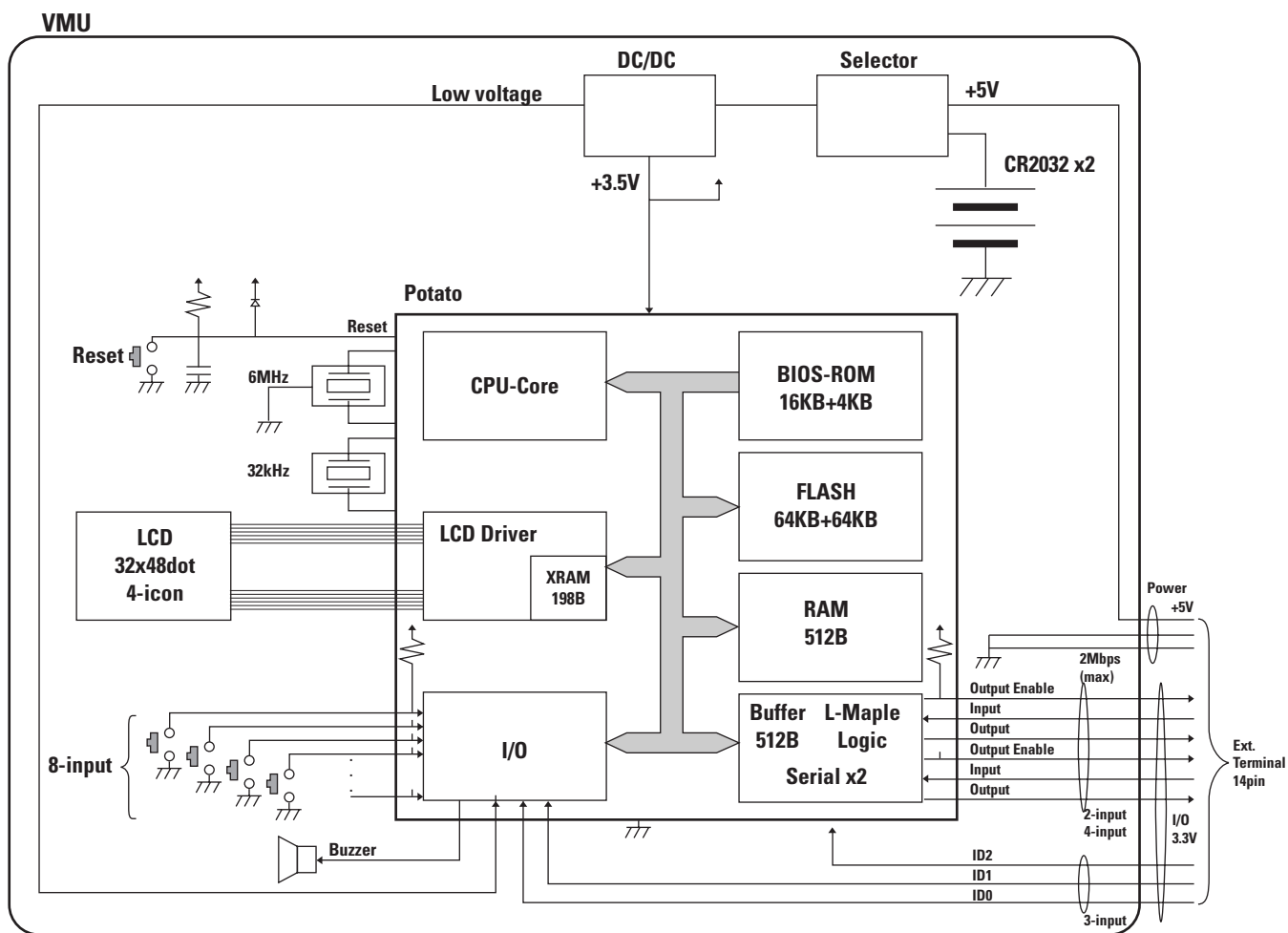
VMU custom chip (Sanyo LC8670)	CPU	8 bit	Instruction cycle time Connected to Dreamcast: 6 MHz (1 ms) Standalone operation: 32 kHz (183 ms) Note: In standalone operation, operation is deliberately slowed down to reduce power consumption.
	Memory	ROM	16 KB system BIOS, system programs
		Flash memory EEPROM	64 KB program/data area 64 KB data area (28 KB reserved for system)
		RAM	256 bytes for applications 256 bytes reserved for system
		Work RAM	512 bytes work RAM. When connected to Dreamcast, reserved by system for use as transfer buffer. In standalone operation, read/write in single byte units possible.
		XRAM (for LCD)	96 bytes for LCD upper half 96 bytes for LCD lower half 6 bytes for icons (reserved by system)
	Serial interface	Used exclusively as follows. (1) Dedicated Dreamcast interface (2) Synchronous 8-bit serial interface with 2 transfer channels	
	Timer	16-bit clock timer 16-bit (or 8-bit x 2), use as PWM sound source possible	
	I/O ports	Input/output 16 lines (buttons, serial interface) 4 lines (control connector)	
LCD controller	33 common lines, 48 segment lines		
LCD	Reflective type liquid crystal	48 (horizontal) x 32 (vertical) dots, 2-value B/W 4 mode icons (file, game, clock, alert), reserved by system	
	Piezoelectric buzzer	Alarm (PWM sound source output)	
Power supply	Button type batter External input External output	CR2032 x 2 +5V, +3.3V input 3.3V output	
Buttons	8 operation buttons + reset button	Direction buttons, A button, B button, MODE button, SLEEP button (reset button)	
Connector	14 pins	Serial interface, power supply	



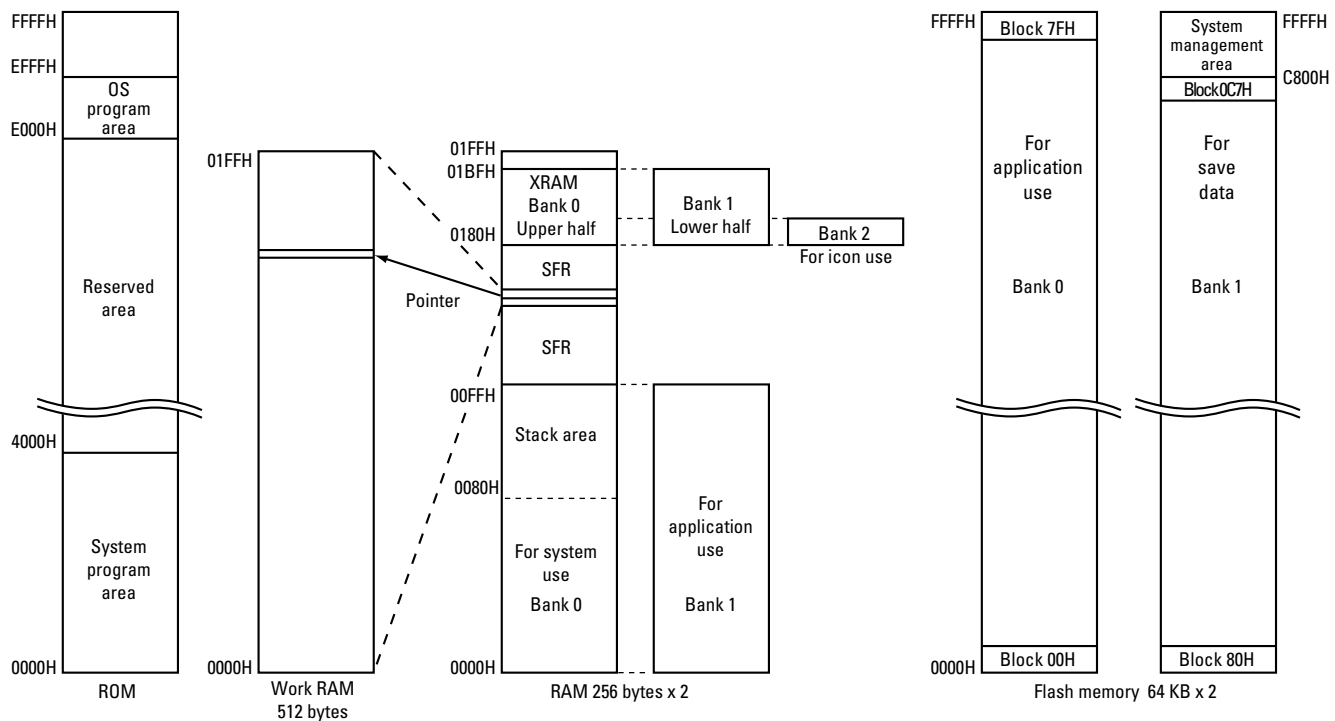
**Figure 2.2** External view



**Figure 2.3** VMU Front View and Rear View



**Figure 2.4** System Block Diagram



**Figure 2.5** VMU Memory Map

## VMU Functions

When connected to the Dreamcast via its dedicated interface, the following functions of the VMU are controlled by the Dreamcast.

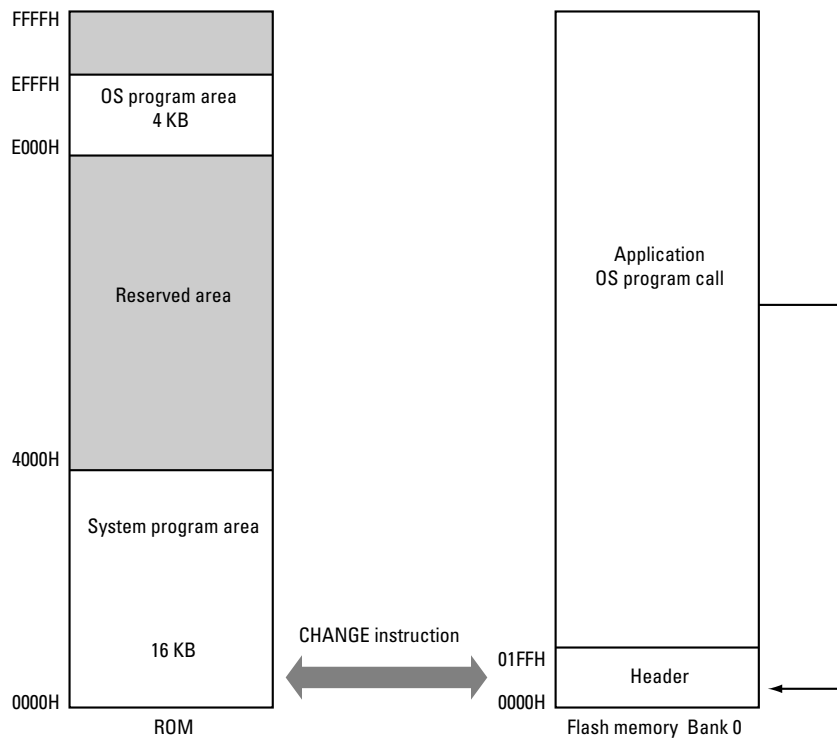
**Note:** The control port of the Dreamcast accepts the standard controller and other devices such as a steering controller etc. Devices which have an expansion device connector can accommodate the VMU or other add-on devices. These peripherals communicate with the Dreamcast via a dedicated bus called the Maple bus.

1. Game data storage medium
2. Controller-specific LCD display
3. VMU clock read and set

These functions are controlled using special programs stored in ROM on the VMU. These programs are collectively referred to as the system BIOS.

The system BIOS consists of system programs, OS programs, and headers. The system programs perform functions such as copying and deleting files, controlling the clock display, and communicating with the Dreamcast. OS programs control basic functions such as flash memory read/write, internal clock setting, battery voltage checking, etc. Some OS programs can be called by applications. For this purpose, a part of the program must be placed in a dedicated location in the flash memory. These parts are called headers.

The VMU contains the following system programs.



**Figure 2.6** Memory Map of Programs in ROM

### File management

File management refers to the handling of game data stored in the Dreamcast and executable files for VMU applications.



Files are managed in units of one block (128 bytes). Reading and writing in block units is possible.

All FAT operations, file name information etc. are handled by system programs.

### Liquid-Crystal Display

When connected to the Dreamcast, the display of the VMU shows only graphics transferred as image data from the Dreamcast. System programs receive the data and handle them for display on the LCD.

In standalone mode, the CPU of the VMU directly controls graphics display. The dot matrix section of the LCD uses a grid of 32 (vertical) x 48 (horizontal) dots. The data amount for one image is 192 bytes. In addition, the LCD also contains four types of icons to indicate operation modes.

Icon	Operation mode	Function
	File mode	VMU file management
	Game mode	Executing game from flash memory
	Clock mode	Date and time display
	Accessing	Flash memory access

Because these icons show the operation mode of the VMU, their status may not be changed by applications.

### Starting VMU applications

A VMU application can be transferred from the Dreamcast and started by the VMU. OS programs also comprise various subroutines that can be used by applications. For details, refer to the "System BIOS" section.

### Data transfer

When connected to the Dreamcast, control of the VMU is performed via a dedicated interface.

In standalone operation, an 8-bit synchronous serial interface is available for communication with another VMU unit.

### Clock

The VMU incorporates a clock which operates at all times, whether connected to the Dreamcast, running an application, or in sleep mode.

Application programs can obtain date and time information using an OS program.

## Buzzer

The piezoelectric buzzer incorporated in the VMU is driven by a pulse generator (PWM) allowing for variable frequencies. In theory, the available frequency range is 21 Hz to 5.5 kHz, with 170 Hz to 2.7 kHz being recommended.

While connected to the Dreamcast, control of the buzzer from the Dreamcast is possible.

During standalone operation, the frequency can be changed by controlling the PWM, and buzzer on/off control is also possible.

## Operation mode switching

The VMU operation mode is determined by the connection method to the Dreamcast and by the MODE button. The current operation mode is indicated by an icon on the LCD.

In standalone operation other than game mode, if no button was pressed or no communication has occurred for more than two minutes, the auto power-off function sets the unit to sleep mode to conserve power.

The VMU has the following operation modes:

- File mode (management of stored game data)
- Game mode (playing a VMU internal game)
- Clock mode (clock display and setting)
- System mode (flash memory access)

**Table 2.2 Dreamcast Connection Status and Operation Mode**

Connection Status	MODE button Icon display	Status	Operation mode
Connected to Dreamcast	Game File Clock	Always out	System mode
	Alert	On	Flash memory access
Standalone operation	Game	On	Application running
	File	On	File operation
	Clock	On	Clock display
	Alert	On	Flash memory access

## Integrated character font

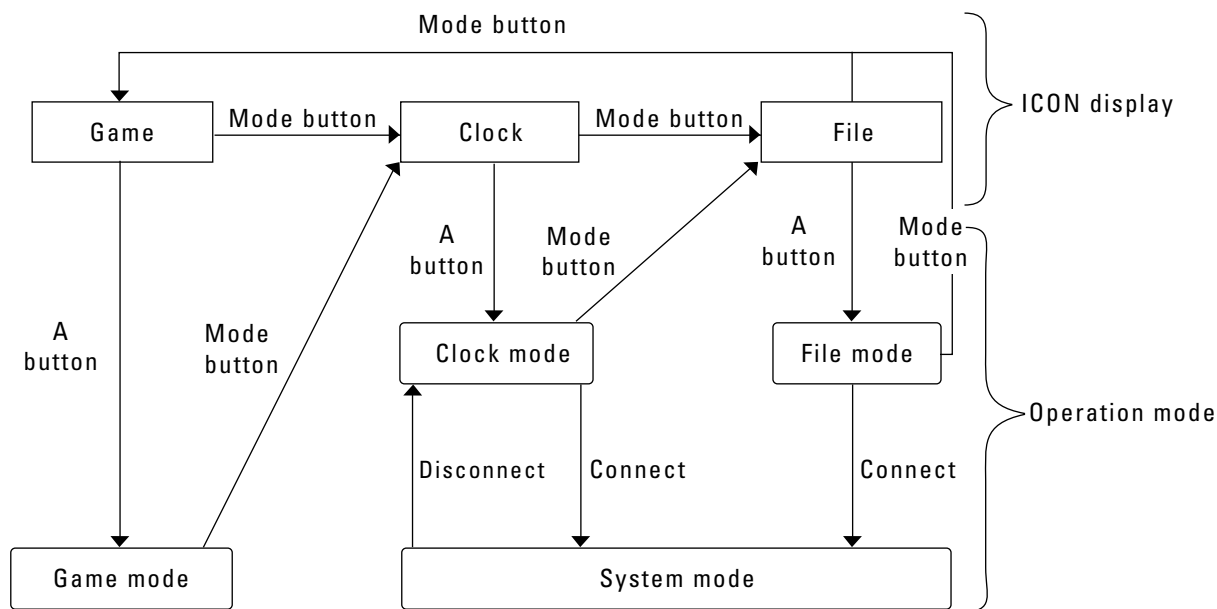
VMU incorporates an ANK font using a grid of 8 (vertical) x 6 (horizontal) dots. The font comprises alphanumeric characters, Japanese katakana, and symbols. The font is only for internal use by the system. It cannot be used by applications, either when connected to the Dreamcast or in standalone mode.

To display characters, image data must be placed in XRAM.



## Mode Setting

The VMU operation mode is determined by the connection method to the Dreamcast and by the MODE button + A button.



**Figure 2.7** Mode Transition

Details of the various operation modes are as follows.

### System mode

VMU is controlled by external control program (Dreamcast). VMU carries out Maple bus compliant communications and memory management, LCD display, and clock management.

When leaving system mode, such as when the VMU is disconnected from the Dreamcast controller, the VMU title screen is shown on the LCD.

### Game mode

In this mode, an application read into the flash memory is executed.

---

**Caution:** Applications should be designed to always check for a MODE button press. When the button is depressed, the application must terminate immediately and control must be handed to the system program. This applies also when the VMU is connected to the Dreamcast controller while an application is running.

---

When a work area in RAM is used, the application should move its contents to flash memory or similar before terminating.

For information on how to terminate applications and hand control to the system program, refer to “Application Shutdown When MODE Button Is Pressed”.

### **File mode**

This mode serves for managing game data stored on the VMU. File management is performed by system programs. The buttons on the VMU are used to display, copy, or delete files written to the flash memory.

### **Clock mode**

In this mode, the time is displayed on the LCD of the VMU. Time can be displayed using hours, minutes, and seconds, and the user can set the time as desired. When connected to the Dreamcast, time setting can also be performed from the Dreamcast side.

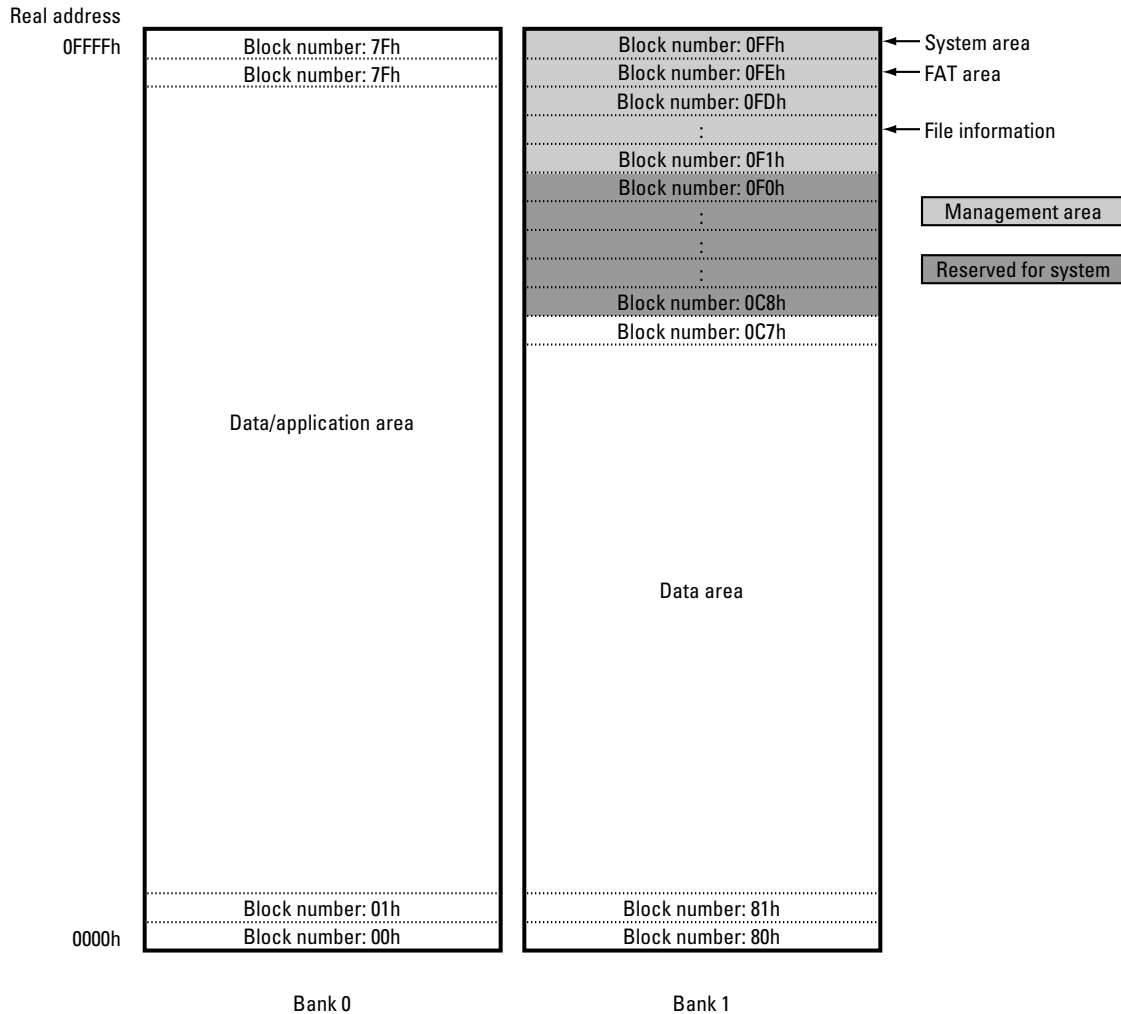
Clock functions are performed using system programs and OS programs.

## File Management

The total capacity of the flash memory on the VMU is 128 KB (64 KB x 2 banks). 28 KB are reserved for the system. Flash memory is managed by the system using 128-byte blocks. The smallest read/write unit for a file therefore is 1 block (= 128 bytes), and up to 200 blocks of data can be stored.

One executable application file can be transferred to the VMU. The executable file must be placed in contiguous blocks starting at block 00H. The maximum size for the executable file is 64 KB (= 128 blocks).

It is not possible to transfer and execute multiple executable files or an executable file larger than 64 KB.



**Figure 2.8** Flash Memory Memory Map

## Flash memory management area

15 blocks starting from the top of the memory range (block 0FFH) are used as memory management area. The management area is divided into the system area (1 block), FAT area (1 block), and file information area (13 blocks).

The system area is write-protected except for VMU formatting performed by the Dreamcast.

The FAT area manages one block using 2 bytes (16 bits), to maintain the block chain configuration.

The file information area holds 32 bytes of information per file and can manage up to 200 files. Out of the 32 bytes, 12 bytes (equivalent to 12 ASCII codes) are used for the file name. Because a hierarchical structure is not supported, subdirectories cannot be created.

### **Data area**

The data area which can hold files consists of 200 blocks extending from block 00H to block 0C7H. Files are placed in this area starting from block 0C7H and going towards block 00H. The application starts from block 00H.

Blocks 00H to 7FH and 80H to 0FFH are managed by bank switching, performed automatically by an OS program.

For reading and writing to the flash memory, always call the OS program.

### **Reserved area**

This area is used by system programs and system modes. Writing to this area is prohibited.

### **LCD Display**

The LCD of the VMU consists of a dot matrix section with 32 (vertical) x 48 (horizontal) dots and an operation mode icon section with 4 icons.

To display images on the LCD, the image data must be stored in the dedicated XRAM.

### **XRAM**

The dedicated RAM used for LCD display is called XRAM. This corresponds to the video RAM in a conventional computer.

The XRAM has 3 banks. Banks 0 and 1 can be written to by applications. Bank 2 serves for operation mode display and cannot be used by applications.

Bank 0 of the XRAM corresponds to the upper half of the LCD (48 x 16 dots), and bank 1 to the lower half (48 x 16 dots).

1 LCD dot corresponds to 1 bit in the XRAM. 1 byte of XRAM controls 8 horizontal dots, with 6 bytes forming one horizontal line.

### **Image mode**

When connected to the Dreamcast, image data received from the Dreamcast are normally written to the XRAM by a system program. However, for display of a secondary game screen, image data are written directly to the VMU. When transferring image data, pay attention to the top/bottom orientation of the VMU. Vertical image reversal can be performed using the Ninja library.

The VMU also incorporates an ANK character font, but this is for exclusive use by system programs. It cannot be used by applications.

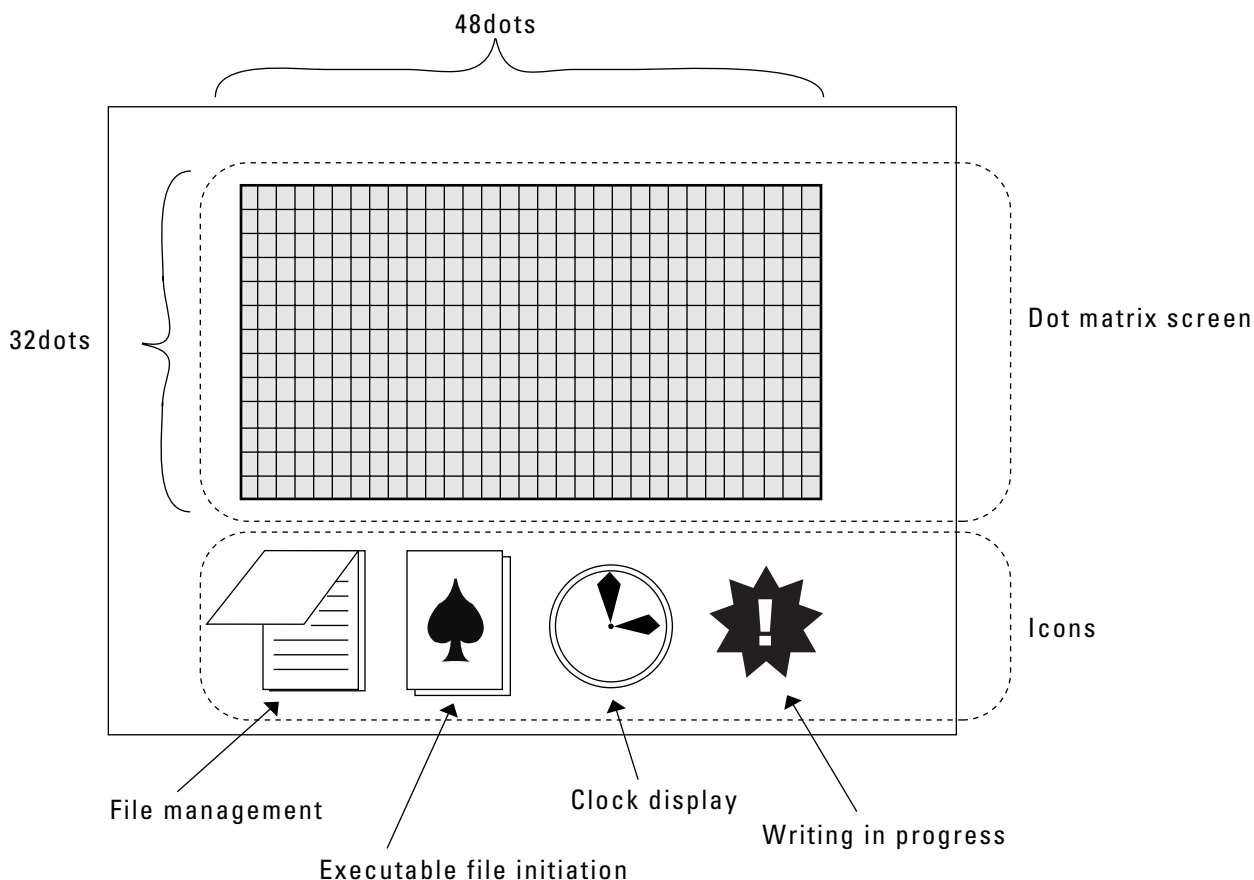
To draw an image on the LCD, XRAM bits for black dots should be set.

### **Icon**

Because the icons show the operation mode of the VMU, their status may not be changed by applications.

### **Image configuration**

The LCD of the VMU is configured as follows.



**Figure 2.9** LCD Screen

## LCD characteristics

The screen refresh principle for an LCD differs from that for a CRT display. After data have been transferred to the XRAM, they are displayed immediately on the LCD, but there is a certain delay due to the response characteristics of the LCD. If this delay is not handled properly, trailing images and flicker will severely impair display quality.

The clock differs in standalone operation and when connected to the Dreamcast. In standalone mode, LCD display speed is slower.

Recommended refresh rate for the LCD of the VMU is 200 ms or more.

## Other important points

Also consider the points listed below when developing applications.

- There is no provision for contrast adjustment (only LCD on/off control)
- There is no provision for brightness adjustment.
- There is no backlight.
- The reflective polarizer plate (rear sheet) of the LCD cannot have a pattern (picture or similar).

# Starting an Executable File

An application can be transferred from the Dreamcast or a conventional computer to the VMU, for execution in standalone mode.

Only one executable file can be transferred to one VMU. It is not possible to use multiple applications simultaneously.

Several OS programs are being made available for use by applications.

## Writing applications for the VMU

Applications for the VMU should be written using an MS-DOS assembler and linker. The conventional executable file created by the linker is converted into an executable file for the VMU by the program E2H86K.EXE.

A VMU application can be debugged using the VMU simulator designed to run under Windows 95 and later. This simulator emulates all aspects of VMU hardware operation in software. For details, refer to the VMU Simulator Guide.

## Transferring an executable file

The executable file is to be stored in blocks 00H to 7fH of the flash memory, starting at block 00H.

Before sending an executable file from the Dreamcast or a conventional computer to the VMU, a contiguous area starting at block 00H must be obtained (defragmented). If the amount of available memory is smaller than the application or if no contiguous area can be obtained, the application cannot be transferred.

For transfer, use the Ninja library and transfer utilities. These allow automatic checking of available space and defragmentation.

## Executable file size

The maximum executable file size is 64K. Larger applications cannot be transferred to the VMU. When an area for storing data in the flash memory is required, this area must be provided for within the application.

---

**Caution:** The executable file comprises the OS program and a program header area containing interrupt vector information. In GHEAD.ASM supplied with the SDK, the program header area is 0000H - 01FFH.

---

## OS programs usable by applications

The following OS programs can be used by applications. When an OS program is called, a part of RAM can be used as work area.

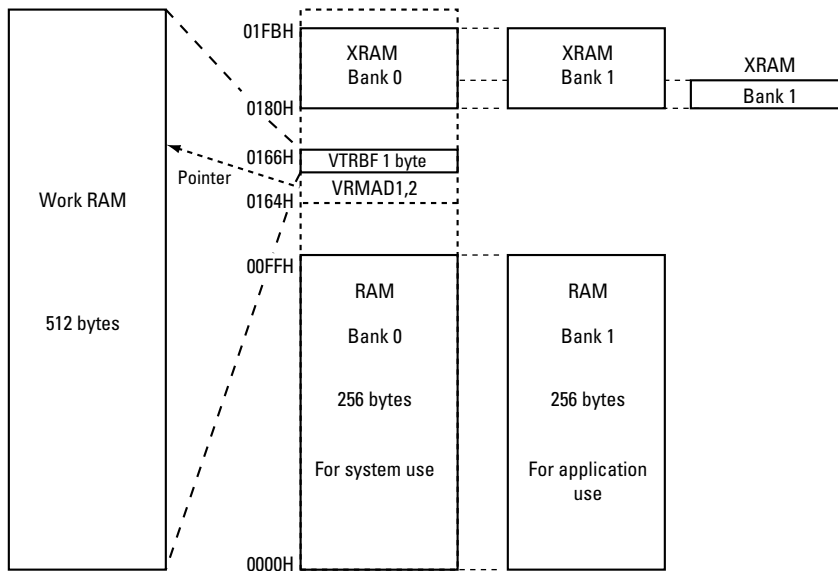
- |                                |  |
|--------------------------------|--|
| 1. Automatic low-battery check | Allows enabling an automatic low battery warning.          |
| 2. Clock read                  | Gets the date and time from the internal clock in the VMU. |
| 3. Flash memory write          | Writes data to flash memory in block units.                |
| 4. Flash memory read           | Reads data from flash memory in block units.               |
| 5. Flash memory verify         | Checks data read from flash memory for validity.           |



## RAM

The following RAM areas are available to applications.

RAM	00H to 0FFH (bank 1) RAM bank 0 is reserved for the system. Except for the stack area, it cannot be used by applications.
Work RAM	00H to 1FFH (read in 1-byte units by specifying address)
XRAM	Bank 0, bank 1

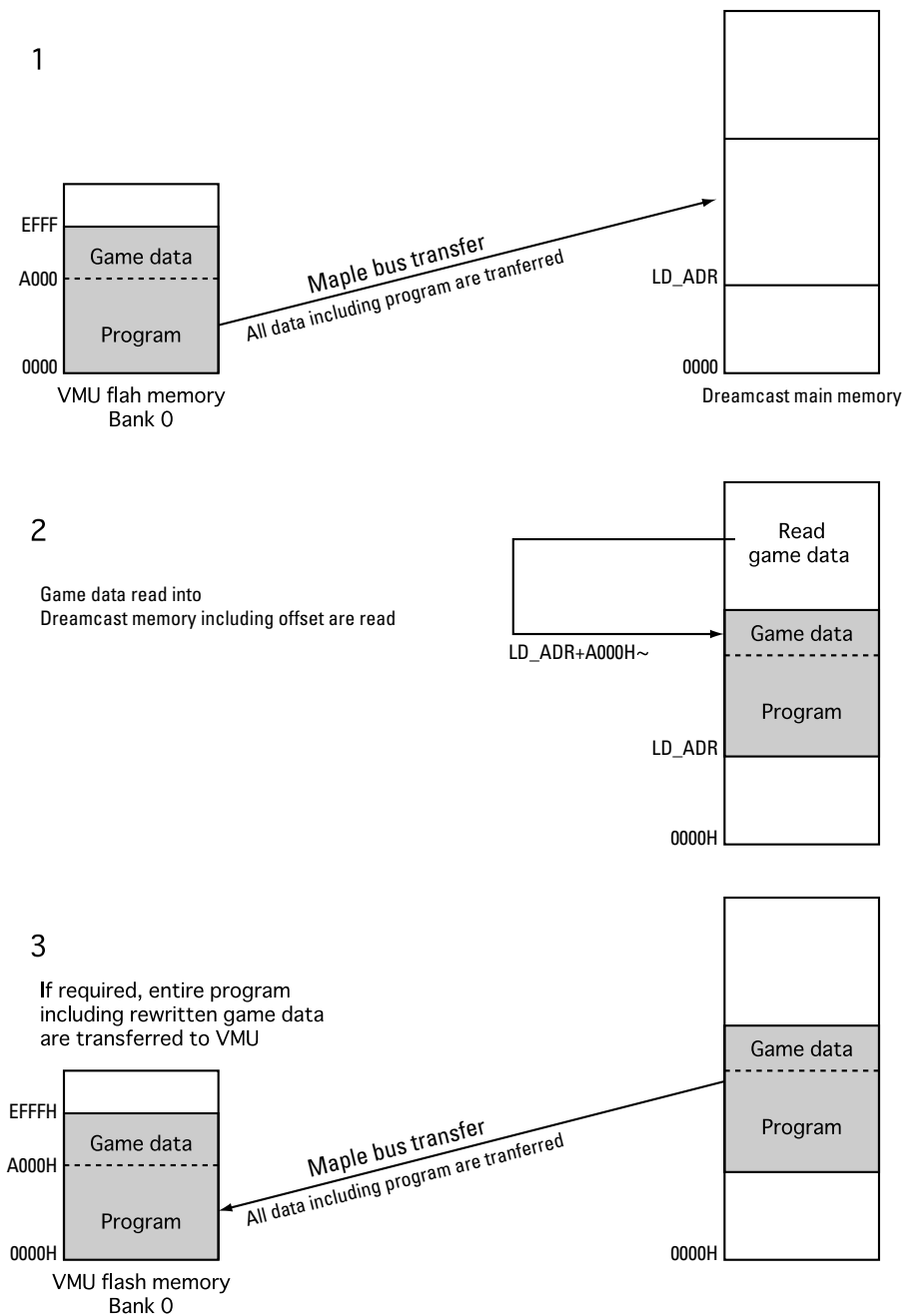


**Figure 2.10** RAM Memory Map

## Saving application data

If a VMU application needs to save progress data or parameters, a data area must be provided within the executable file. Because the executable file is read into flash memory, the data area also will be stored in flash memory. It is not possible to create files of the same format as for Dreamcast save data.

When data saved in a VMU application are to be used as Dreamcast applications or links, read the entire VMU application and perform a lookup on the data addresses in it.



**Figure 2.11** Linking of Save Data in VMU and Dreamcast

## Auto power-off

The VMU incorporates an auto power-off function that automatically sets the unit to sleep mode if no button was pressed or no communication has occurred for more than two minutes. For details on the sleep mode of the VMU, refer to section "Sleep Mode".

When game mode is active, auto power-off is disabled. Applications must provide their own sleep mode. For details, refer to section "Sleep Mode".

## **Communication Functions**

VMU can communicate with other devices via a serial interface. Two protocols are available. When connected to the Dreamcast, the Maple bus protocol is used. In standalone operation, full-duplex synchronous serial transfer is used. Protocol switching is performed automatically by a system program detecting the Dreamcast connection status.

### **Maple bus protocol**

When connected to the Dreamcast, the communication connector of the VMU becomes a 2 Mbps Maple bus connector. The entire work RAM is used as send / receive buffer. If an application was using the work RAM as work area, the entire contents will be destroyed. Take this into account when designing applications.

The Maple bus cannot be used by applications.

### **Synchronous serial transfer**

When the VMU is operating in standalone mode and data transfer is carried out between two VMU units or between one VMU and a computer, synchronous serial transfer is used. There are two serial communication lines, allowing full-duplex operation. Data can be transferred in 1 byte units, and the maximum transfer rate is about 2.4 kbps.

## **Clock Function**

A 32.768 kHz quartz oscillator and dedicated counter keep time in 500-millisecond units. Date and time data are managed by an OS program. These data can be read by an application, but not written to.

A setting screen is used to set the year, month, day, and time. When connected to the Dreamcast, the clock in the VMU can be set from the Dreamcast.

## **Alarm Function**

The piezoelectric buzzer incorporated in the VMU can be used for an alarm. The buzzer can emit a single tone at a time. In theory, the available frequency range is 21 Hz to 5.5 kHz, with 170 Hz to 2.7 kHz being recommended. The alarm function can be implemented by setting the timer (pulse generator) connected to the buzzer. The buzzer can be switched on and off, but volume control is not possible.

When connected to the Dreamcast, the Dreamcast can control the buzzer of the VMU, including the frequency setting.

# **Sleep Function**

The VMU incorporates a sleep function designed to conserve power when operating in standalone mode. In the sleep condition, the state of the I/O ports and the contents of RAM are maintained, but the CPU and LCD are turned off.

In game mode, transition to the sleep state is controlled by the application. In clock mode and file mode, the following conditions cause transition to sleep mode.

- SLEEP button was pressed
- Auto power-off function was activated
- No button press or communication for about 2 minutes

To cancel the sleep mode, the SLEEP button must be pressed. Other buttons are disregarded.

The RAM and register memory contents are preserved, except for the clock register.

## Buttons

The VMU has the following buttons.

Applications should be designed so as to maintain the interface described below.

Button name	Main function
Direction button (up)	Cursor movement and display scrolling
Direction button (down)	Cursor movement and display scrolling
Direction button (left)	Cursor movement and display scrolling
Direction button (right)	Cursor movement and display scrolling
A button	Mainly "confirm"
B button	Mainly "cancel"
MODE button	Mode switching during standalone operation Each push cycles through "File" → "Game" → 'Clock' → "File"...
SLEEP button	Activating and canceling of sleep mode in standalone operation
Reset button	Reset of unit contents except flash memory and clock

### **Batteries**

The VMU incorporates two button-size batteries (CR2032) which act as a power source in standalone operation. While connected to the Dreamcast, power is supplied by the Dreamcast.

Battery life will depend on the usage conditions of applications. Under the conditions outlined below, the batteries will last about two weeks.

- VMU standalone operation
- LCD display on (refresh rate 1 kHz)
- No alarm output
- Communication functions not used
- No write to flash memory

For specific information on how to calculate expected battery life for an application, refer to section "Calculating Battery Life".



## Battery life

Battery life depends on the operation condition of the VMU. Refer to the table below to calculate battery life for applications.

Operation	Battery power consumption	Comments
Program running	Standard	Reference for flash memory read and CPU battery power consumption
LCD screen update	Standard x5	Frequent XRAM rewriting (screen update) consumes battery power
Alarm output	Slightly more than standard	Slight increase in battery power consumption
Flash memory write	Standard x25	Flash memory writes should be limited to minimum because of extremely high power consumption
Data transfer		Very high battery consumption, especially by applications which also write to flash memory. When transferring large files, take battery life into consideration.

## Battery status monitoring

An OS program continuously monitors the battery voltage. When the batteries near the end of their service life, the program will trigger auto power-off, even if an application is running.

## Battery replacement

When the batteries are replaced, the clock will be initialized, but the contents of flash memory are not affected.



## ***CPU Features***

---

The VMU is a memory system for the Dreamcast game machine. The CPU in the custom LSI chip has a minimum cycle time of 0.5 ms. Other functions integrated on this chip are a 128- KB flash memory, 20-KB ROM, 710-byte RAM, LCD controller/driver, 16-bit timer/counter/pulse generator, 16-bit (or 2-channel x 8-bit) timer, 2-channel x 8-bit synchronous serial interface, dedicated Dreamcast interface, and 13-source, 10-vector interrupt architecture.

## Differences to Conventional CPUs

Normally, a CPU will have an internal accumulator as well as general registers and flag registers. The control registers and data registers for the serial port and other peripheral devices are mapped onto the I/O ports.

In the VMU custom chip, all CPU and peripheral device registers are mapped onto memory. These registers are referred to as “special function registers” (SFR) and are treated separately from RAM.

Keep in mind that these “special function registers” are not internal registers of the CPU.

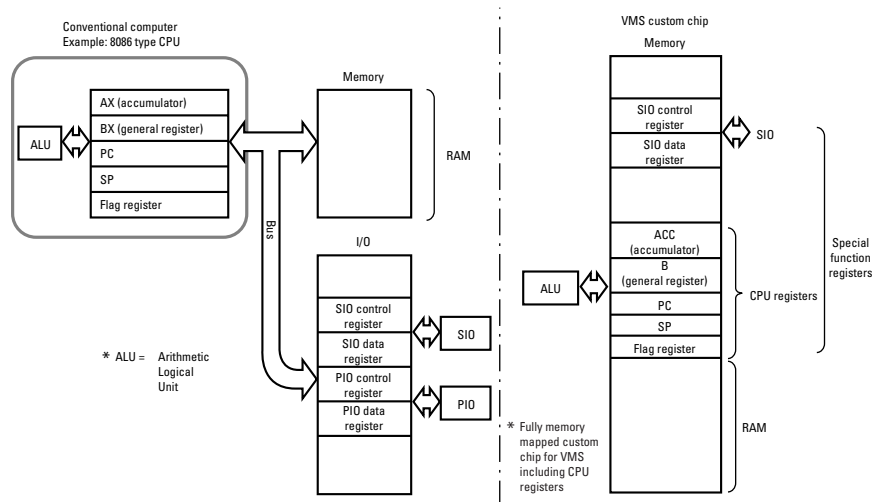


Figure 2.12 Differences to Conventional CPU

## Specifications

This section gives an overview of VMU specifications.

### Memory specifications

#### Flash memory

65536 bytes: Program/ data area

65536 bytes: Data area

#### ROM

16384 bytes: Program area

4096 bytes: System BIOS program area

#### RAM

Arithmetic area: 256 bytes x 2 banks

Display area: 198 bytes (LCD video XRAM)

Work area: 256 bytes x 2 banks (work RAM)

---

**Note:** The work RAM in the work area is used as a send/receive buffer when connected to the Dreamcast.

---

### Bus cycle time and instruction cycle time

The bus cycle time refers to the ROM read time.

Bus cycle time	Instruction cycle time	System clock oscillator	Oscillation frequency	Power supply voltage	Others
3.412ms	6.824ms	RC oscillator	879.236kHz	3.15 to 3.8V	OCR7=1*1
91.553ms	183.105ms	Quartz oscillator	32.768kHz	3.15 to 3.8V	OCR7=1*1

---

**Caution:** OCR7 (bit 7 of the oscillation control register OCR) controls the system clock generator operation and cycle time. For details, refer to section “System Clock Generator”.  
 OCR7 = 1: 1/6 of system clock is used as cycle time  
 The frequency of the RC oscillator circuit is subject to tolerances. The reference value is 879.236 kHz, but the frequency can range from about 600 kHz to 1200 kHz.

---

### **Ports**

I/O ports: 2 (P1, P3)

Input port: 1 (P7)

LCD segment drive output ports: 48

LCD drive common output ports: 33

### **LCD controller**

Display duty cycle: 1/33

Display bias: 1/5

LCD instruction: on/off

Graphics display: 32 vertical x 48 horizontal dots + 4 icons

### **Serial interface**

8-bit serial interface x 2 channels (synchronous)

Integrated 8-bit baud rate generator (also used for 2- channel serial interface)

Dedicated Dreamcast interface (automatic start patten/end pattern detection)

---

**Caution:** Synchronous serial interface and dedicated Dreamcast interface cannot be used simultaneously.

---

### **Timer**

#### **Timer 0**

16-bit timer/counter

with 8-bit programmable prescaler

#### **Timer 1**

16-bit timer/pulse generator

Base timer: clock selector function

Selects between 32.768 kHz quartz oscillator, system clock, timer 0 programmable prescaler output

500-ms overflow signal generator for clock (when 32.768 kHz quartz oscillator is selected)

Overflow signal generator for 976 ms, 3.9 ms, 15.6 ms, or 62.5 ms cycle (when 32.768 kHz quartz oscillator is selected)

## Interrupts

The interrupt architecture comprises 13 sources and 10 vectors

- 1) External interrupt INT0: connection detection for dedicated Dreamcast interface
- 2) External interrupt INT1: low power supply voltage interrupt
- 3) External interrupt INT2: timer / counter T0L (timer 0, lower 8 bits)
- 4) External interrupt INT3: base timer
- 5) Timer / counter T0H (timer 0, upper 8 bits)
- 6) Timer T1L (lower 8 bits), timer T1H (upper 8 bits)
- 7) Serial interface 0 (SIO0)
- 8) Serial interface 1 (SIO1)
- 9) Dedicated Dreamcast interface
- 10) Port 3

---

**Caution:** The clock function of the VMU is implemented by counting the interrupts generated in 0.5 second intervals by the base timer. The port 3 interrupt is a level interrupt which is maintained for as long as the user presses a button. If the timer is used to frequently generate interrupts or to accept the port 3 level interrupt, the internal clock may run slow. When using the base timer interrupt, call the user-side handler immediately after the label `timer_ex_exit` in `GHEAD.ASM`. The user-side handler must be designed to keep processing time at a minimum, so that the interrupt can be properly processed every 0.5 seconds. Care must be taken to prevent clock slow-down already when designing an application.

---

Priority can be assigned to the interrupts using three interrupt levels (low, high, top). The interrupt priority register can be used to specify high or low priority for the 11 interrupt sources of port 3 for external interrupt INT2 and timer / counter T0L (timer 0, lower 8 bits). High or low priority can also be specified for external interrupt INT0 and INT1.

### **Stack area**

128 bytes in RAM bank 0, from 80H to 0FFH. The internal clock uses 20 bytes. The stack is used up from the top (80H).

### **High-speed arithmetic instructions**

16 bit x 8 bit (execution time: 7 command cycles)

16 bit  $\otimes$  8 bit (execution time: 7 command cycles)

### **3 oscillator circuits**

RC oscillator: system clock (reference: 879.236 kHz; tolerance range: 600 to 1200 kHz)

Quartz oscillator: clock, system clock, LCD driver clock (32.768 kHz)

### **Standby function**

Sets CPU to HALT mode. In this mode, instructions are not executed, but the internal clock continues to operate. The mode can be canceled by a reset or interrupt.

The mode is identical to the sleep mode of the VMU, which can be canceled by pressing the SLEEP button.

### **Flash memory specifications**

Memory type: EEPROM (Electrically Erasable Programmable ROM)

Capacity: 128 KB

Write method: using OS program

Write block size: 128 KB

Erase/write voltage: 3.15 to 3.8 V

Maximum number rewrite cycles: 50,000 (each cycle consisting of one FFH write and 00H write operation)

Ta = 25 °C, memory managed by program

Program memory space: 64 KB

System BIOS (ROM)/Application (flash memory) switching: by CHANGE instruction. At reset, BIOS is activated.



## System block diagram

A block diagram of the VMU is shown below.

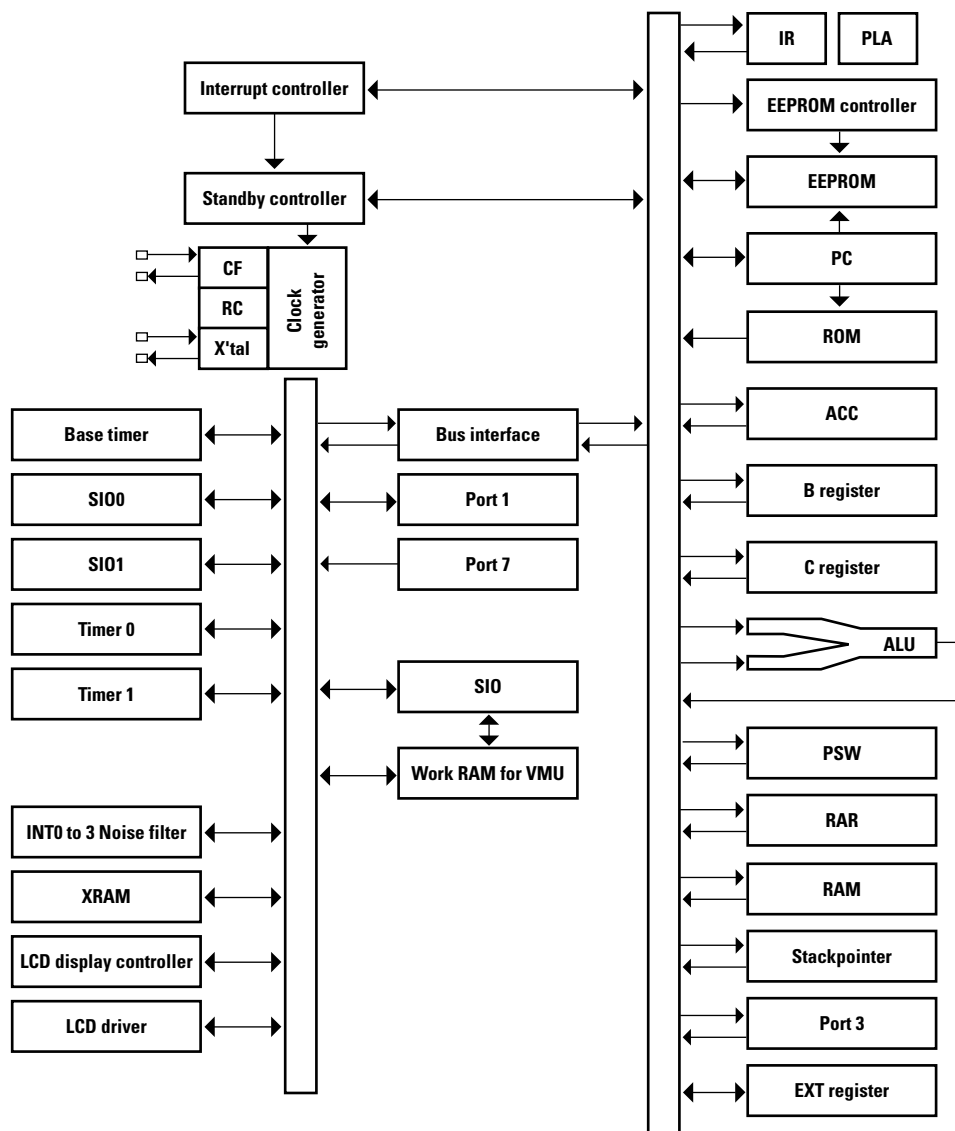


Figure 2.13 VMU System Block Diagram



---

# ***Internal System Configuration***

---

Unlike in a conventional CPU, the accumulator and all registers are mapped to RAM. The relationship between CPU functions and special function registers is described in this section.

## **Memory Space**

The VMU custom chip comprises internal memory space and flash memory space. The internal memory space is divided into ROM (64 KB) and RAM (512 bytes). In ROM, sequential addresses are incremented with each normal instruction execution, allowing linear access to 64 KB.

In RAM, the 256 bytes formed by address range 000 to 0FFH are assigned as general-purpose RAM. The 256 bytes formed by address range 100 to 1FFH are assigned to the special function registers (SFR). General-purpose RAM consists of 2 banks. The bank can be specified by bit 1 (RAMBK0) of the program status word (PSW) of the special function registers (SFR). Bank 0 is also used as stack area. The SFR comprises accumulator (ACC), PSW, timer, I/O ports etc., forming a completely memory-mapped I/O configuration.

The flash memory space has a capacity of 128 KB, divided into 2 banks of 64 KB each. Bank 0 only is available for execution of application programs. Switching between the ROM system BIOS and a program in flash memory is performed by a dedicated macro instruction (CHANGE). Data writing to flash memory must be performed by calling the appropriate OS program.

---

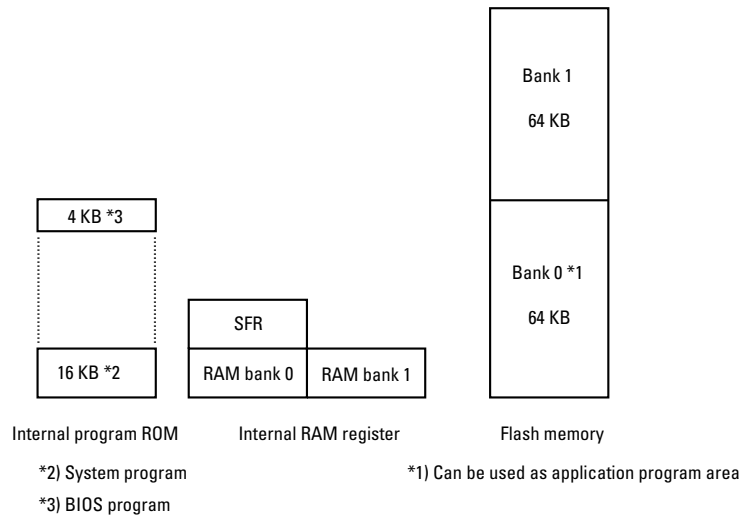
**Caution:** When accessing the flash memory, inhibit all interrupts including the base timer. Because the base timer is used by the internal clock, the inhibit interval should be kept as short as possible.

---

For writing to the flash memory, set the system clock to RC oscillator and the division ratio to 1/6. For write and verify, set the system clock to RC oscillator and the division ratio to 1/12.

OS program routines in ROM are provided for flash memory write, data verify, and read operations.

A VMU application always is stored in bank 0 of the flash memory.



**Figure 2.14** *Three Memory Space Types*

## Program Counter (PC)

The program counter (PC) uses a 16-bit configuration for storing the address of the program memory (ROM) where the next instruction to be executed is stored. The CPU refers to the PC value to execute a series of program instructions. The PC is normally incremented in steps of one instruction. When divider instructions and subroutines are executed and when interrupt or reset requests are processed, values for the respective operation states are set in the PC. These values are shown in the table below.

**Table 2.3 Program Counter Setting Values**

Operation			Program counter value
Reset			0000H (internal program space)
External interrupt 0			0003H
External interrupt 1			000BH
External interrupt 2, timer/counter TOL interrupt			0013H
External interrupt 3, base timer interrupt			001BH
Timer/counter TOH interrupt			0023H
Timer T1L, timer T1H interrupt			002BH
SIO0 interrupt			0033H
SIO1 interrupt			003BH
VMU SIO interrupt			0043H
Port 3 interrupt			004BH
Unconditional branch instruction	JMP	a12	PC15 to PC12 = current page PC11 to 00=a12
	JMPF	a16	C15 to 00=a16
	BR	r16	(PC+2)+r8 [128 to +127]
	BRF	r16	(PC+2)+r16 [0 to +65535]
Conditional branch instruction	BZ_ BNZ_ BP_ BNE BPC_ BN_ DBNZ_ BE		(PC+2 or +3) +r8 [-128 to +127]
	CALL instruction	CALL	C15 to C12 = current page PC11 to 00=a12
	CALLF	a12	C15 to 00=a16
	CALLR	16	(PC+2)+r16 [0 to +65535]
Macro instruction	CHANGE label name (or address)		Value specified by other program mode label or address

**Caution:** For convenience, 4 KB of ROM space are referred to as a page. The "current page" refers to the page which contains the instruction that is to be executed after the currently running instruction. When an interrupt is generated during ROM program execution, the interrupt vector in ROM (address in above table) is called. When an interrupt is generated while an application in flash memory is executing, the interrupt vector of bank 0 in flash memory (address in above table) is called. Applications cannot arbitrarily specify interrupt vectors. Rather, the specified program must be included in the application. For details, refer to section 5.1 "Interrupt Functions".

## ROM Space

The 64 KB ROM space comprises 16 KB for system programs and 4 KB for OS programs.

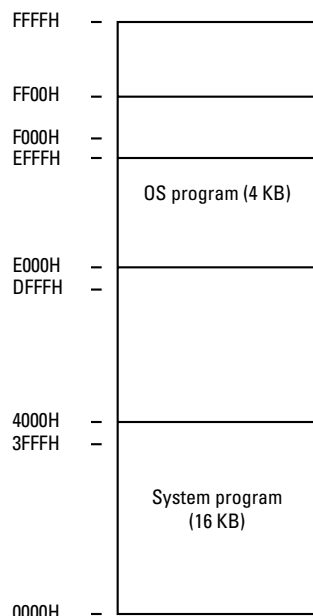


Figure 2.15 ROM Space

## RAM Space

1222 bytes of RAM are included, comprising 198 bytes of LCD video XRAM and 512 bytes VTRBF work RAM. The special function registers (SFR) are located in the top address range (100H to 1FFH) of RAM.

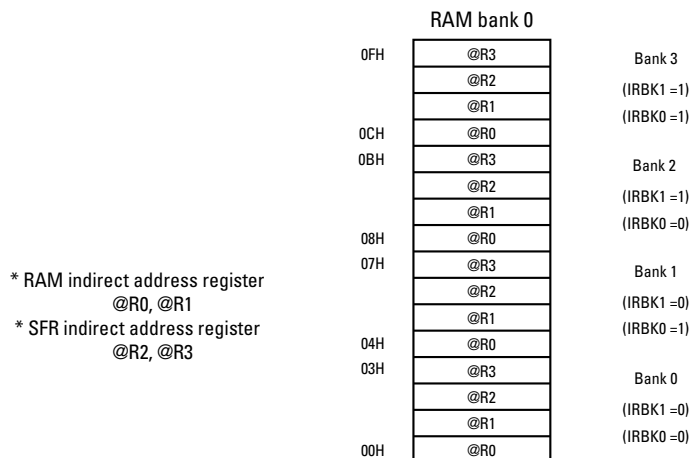
Table 2.4 RAM configuration

Memory	Capacity
RAM size	1222 bytes
XRAM	Bank 0 180H - 1FBH (96 bytes)
	Bank 1 180H - 1FBH (96 bytes)
	Bank 2 180H - 185H (6 bytes)
Main RAM	Bank 0 000H - 0FFH (256 bytes)
	Bank 1 000H - 0FFH (256 bytes)
VTRBF	166H (256 bytes x 2 banks)

## Indirect Address Registers

The 16-byte address range 00H to 0FH in RAM contains 4 banks of indirect address registers. Starting from the lowest address, these consist of @R0, @R1 (for RAM), @R2, @R3 (for SFR). For addressing, the indirect address register banks are specified by bits 3 and 4 of the program status word (PSW) (indirect address register bank flag: IRBK0, 1). This 16-byte area can also be used as regular RAM.

The relationship between indirect address registers and RAM is shown in the table below.



**Figure 2.16** Indirect Address Register Arrangement

**Table 2.5** Indirect Address Register Map

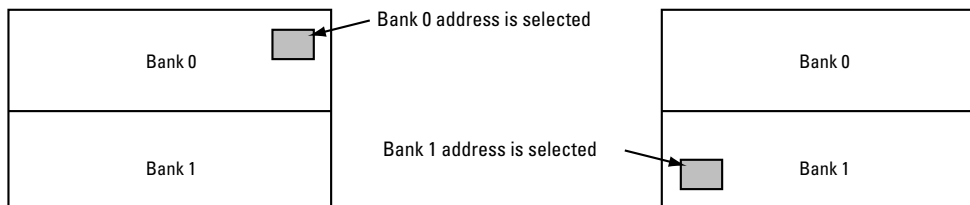
Indirect address register name	Function	Bank 0 (IRBK1=0) (IRBK0=0)	Bank 1 (IRBK1=0) (IRBK0=1)	Bank 2 (IRBK1=1) (IRBK0=0)	Bank 3 (IRBK1=1) (IRBK0=1)
_R0	RAM access	RAM 00H	RAM 04H	RAM 08H	RAM 0CH
_R1	RAM access	RAM 01H	RAM 05H	RAM 09H	RAM 0DH
_R2	SFR access	RAM 02H	RAM 06H	RAM 0AH	RAM 0EH
_R3	SFR access	RAM 03H	RAM 07H	RAM 0BH	RAM 0FH

### (1) Direct addressing mode

When executing instructions such as MOV #i8, d8

(1) RAM bank 0 (PSW 21 = 0)

(2) RAM bank 1 (PSW 21 = 1)

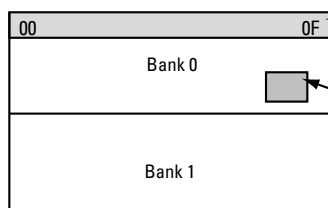


**Figure 2.17** Direct Addressing Mode

## (2) Indirect addressing mode

When executing instructions such as MOV #i8, @Rj

(1) RAM bank 0 (PSW 21 = 0)



(2) RAM bank 1 (PSW 21 = 1)

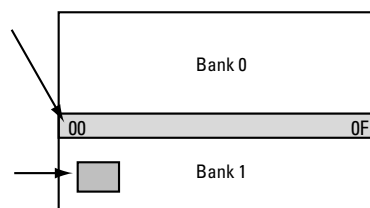


Figure 2.18 Indirect Addressing Mode

## Special function registers (SFR)

A table of RAM and SFR is shown in Table below. For information on the various registers in the SFR range, refer to the sections on the various items.

**Caution:** The initial values are the values established by the BIOS after a reset.

R = READ X = Undetermined  
W = WRITE H = Does not exist

Table 2.6 RAM Memory Map

Symbol	Address	R/W	Designation	Default value	See page
RAM (bank 0)	000H-0FFH	R/W	Data memory	XXXXXXXX (stored at reset)	43
RAM (bank 1)	000H-0FFH	R/W	Data memory	XXXXXXXX (stored at reset)	43
ACC	100H	R/W	Accumulator	00000000	50
PSW	101H	R/W	Program status word	00H00000	52
B	102H	R/W	B register	00000000	51
C	103H	R/W	C register	00000000	51
TRL	104H	R/W	Table reference register lower byte	00000000	54
TRH	105H	R/W	Table reference register upper byte	00000000	54
SP	106H	R/W	Stack pointer	XXXXXXXX	53
PCON	107H	R/W	Power control register	HHHHHH00	158
IE	108H	R/W	Master interrupt enable control register	0HHHHH00	138
IP	109H	R/W	Interrupt priority control register	00000000	151
EXT	10DH	R/W	External memory control register	HHHH0000	—



## Internal System Configuration

OCR	10EH	R/W	Oscillation control register	0H00HH00	156
TOCNT	110H	R/W	Timer 0 control register	00000000	67
TOPRR	111H	R/W	Timer 0 prescaler data	00000000	71
TOL	112H	R	Timer 0 low	00000000	71
TOLR	113H	R/W	Timer 0 low reload data	00000000	71
TOH	114H	R	Timer 0 high	00000000	72
TOHR	115H	R/W	Timer 0 high reload data	00000000	72
T1CNT	118H	R/W	Timer 1 control register	00000000	83
T1LC	11AH	R/W	Timer 1 low comparison data	00000000	86
T1L	11BH	R	Timer 1 low	00000000	85
T1LR		W	Timer 1 low reload data	00000000	85
T1HC	11CH	R/W	Timer 1 high comparison data	00000000	87
T1H	11DH	R	Timer 1 high	00000000	86
T1HR		W	Timer 1 high reload data	00000000	86
MCR	120H	W	Mode control register	00000000	127
STAD	122H	R/W	Start address register	00000000	129
CNR	123H	W	Character count register	H0000000	130
TDR	124H	W	Time division register	HH000000	130
XBNK	125H	R/W	Bank address register	HHHHHH00	130
VCCR	127H	W	LCD contrast control register	00000000	131
SCON0	130H	R/W	SI00 control register	00H00000	108
SBUF0	131H	R/W	SI00 buffer	00000000	113
SBR	132H	R/W	SI00 baud rate generator	00000000	113
SCON1	134H	R/W	SI01 control register	00000000	111
SBUF1	135H	R/W	SI01 buffer	00000000	113
P1	144H	R/W	Port 1 latch	00000000	58
P1DDR	145H	W	Port 1 data direction register	00000000	58
P1FCR	146H	W	Port 1 function control register	10111111	59
P3DDR	14DH	W	Port 3 data direction register	00000000	62
P3INT	14EH	R/W	Port 3 interrupt function control register	11111101	62
P7	15CH	R	Port 7 latch	HHHHXXXX	64
I01CR	15DH	R/W	External interrupt 0, 1 control	00000000	135

## Visual Memory Unit (VMU) Hardware Manual

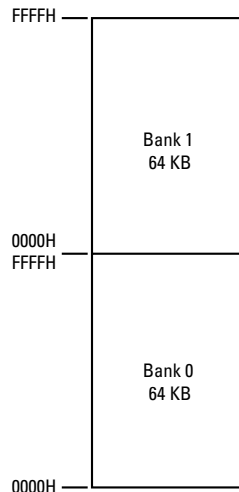
---

I23CR	15EH	R/W	External interrupt 2, 3 control	00000000	137
ISL	15FH	R/W	Input signal select	11000000	138
VSEL	163H	R/W	Control register	11111100	143
VRMAD1	164H	R/W	System address register 1	00000000	144
VRMAD2	165H	R/W	System address register 2	HHHHHHH0	144
VTRBF	166H	R/W	Send/receive buffer	XXXXXXXX	144
BTCR	17FH	R/W	Base timer control	01000001	101
RAM (XRAM) (Bank 0)	180H-1FBH	R/W	LCD memory	XXXXXXXX (stored at reset)	126
RAM (XRAM) (Bank 1)	180H-1FBH	R/W			
RAM (XRAM) (Bank 2)	180H-185H	R/W			

## Flash Memory

The VMU custom chip comprises a 128 KB flash memory space which consists of two 64-KB banks. Reading and writing data from and to the flash memory is performed by calling the appropriate OS program. By using the ROM table lookup instruction (LDC), ROM space data can be accessed. Applications are always placed in the 64 KB memory space of bank 0. Switching between the system BIOS (ROM) and an application (flash memory) is performed by a dedicated macro instruction (CHANGE).

Flash memory size: 64 KB x 2 banks  
 Banks: Bank 0, bank 1  
 Bank address: 0000H - FFFFH



**Figure 2.19** Flash Memory Map

Data read/write for the flash memory is performed by calling an OS program. For details, refer to chapter 12 “Subroutine Reference” in the System BIOS manual.

## Accumulator

The accumulator (ACC) is an 8-bit register used for data arithmetic processing, transfer, I/O operations etc. It is assigned to address 100H of SFR, and initialized to 00H after a reset.

Unlike in a conventional CPU, a part of the memory is used to serve as accumulator.

**Table 2.7** Accumulator (ACC)

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ACC	100H	R/W	ACC7	ACC6	ACC5	ACC4	ACC3	ACC2	ACC1	ACC0
Reset			0	0	0	0	0	0	0	0

## B Register, C Register

The B register and C register are 8-bit registers used in combination with the accumulator for arithmetic operations. They are assigned to address 102H (B register) and address 103H (C register) of SFR, and initialized to 00H after a reset.

## B register

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
B	102H	R/W	B7	B 6	B 5	B 4	B 3	B 2	B 1	B 0
Reset			0	0	0	0	0	0	0	0

## C register

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
C	103H	R/W	C7	C6	C5	C4	C3	C2	C1	C0
Reset			0	0	0	0	0	0	0	0

Multiplication is performed using 16 bits  $\times$  8 bits. For the multiplicand (16 bits), the upper 8 bits are stored in the accumulator and the lower 8 bits in the C register. The multiplier (8 bits) is stored in the B register. The processing result (product) has 24 bits. The top 8 bits are stored in the B register, middle 8 bits in the accumulator, and lower 8 bits in the C register. Therefore, the following applies:

$$(ACC) (C) \times (B) = (B) (ACC) (C)$$

Division is performed using 16 bits  $\div$  8 bits. For the dividend (16 bits), the upper 8 bits are stored in the accumulator and the lower 8 bits in the C register. The divisor (8 bits) is stored in the B register. The processing result (quotient) has 16 bits. The upper 8 bits are stored in the accumulator, and the lower 8 bits in the C register. The surplus is stored in the B register. Therefore, the following applies:

$$(ACC) (C) \div (B) = (B) (ACC) (C) \text{ mod } (B)$$

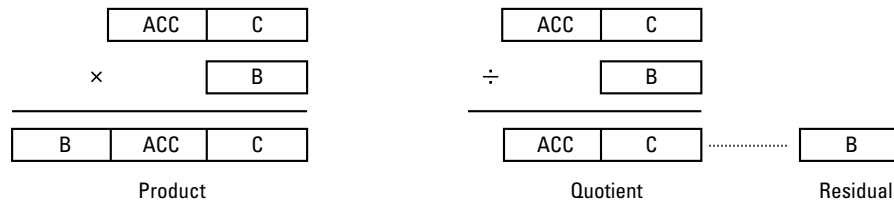


Figure 2.20 Arithmetic Register Contents

## Program Status Word (PSW)

The program status word (PSW) consists of flags indicating the arithmetic processing result status and flags specifying the RAM banks and indirect address registers. It is assigned to address 101H of SFR, and initialized to 0 after a reset.

Table 2.8 Program status word (PSW)

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PSW	101H	R/W	CY	AC	—	IRBK1	IRBK0	OV	RAMBK0	P
Reset			0	0	0	0	0	0	0	0

**CY (bit 7): carry flag**

CY is set (1) when the processing result carries over to the next higher digit (carry) or to the next lower digit for subtraction and comparison (borrow). Otherwise the flag is reset (0). The flag is influenced by rotating instructions that include CY, and is reset (0) when an arithmetic instruction is executed.

**AC (bit 6): auxiliary carry flag**

AC is set (1) when the ACC bit 3 carries over to the next higher digit (carry) or to the next lower digit (borrow). Otherwise the flag is reset (0).

**IRBKx: indirect address register bank flag**

Consists of IRBK1 (bit 4) and IRBK0 (bit 3) which specify indirect address register bank flag 1 and indirect address register bank flag 0.

Serve for specifying the 4 register banks used as indirect address registers for indirect addressing within each RAM bank.

Bank	IRBK1	IRBK0
0	0	0
1	0	1
2	1	0
3	1	1

**OV (bit 2): overflow flag**

When overflow occurs, the OV bit is set (1). Otherwise it is reset (0). This means that the bit is set when the result of an arithmetic operation involving "negative number" + "negative number" or "negative number" - "positive number" is positive, or when the result of an arithmetic operation involving "positive number" + "positive number" or "positive number" - "negative number" is negative. For multiplication and division, the bit is set when the contents of the B register are not 0, and reset when the contents of the B register are 0.

**RAMBK0 (bit 1): RAM bank flag**

Serves for specifying the RAM bank. When an instruction performs RAM access, the RAM address within the specified bank is accessed.

Bank	RAMBK0
0	0
1	1

**P (bit 0): accumulator (ACC) parity flag**

When the total number of bits set in the accumulator is odd, this bit is set (1). When the number is even, the bit is reset (0). This bit is read-only.

## Stack Pointer

RAM bank 0 is used as stack memory. The 8-bit SP register is used to specify addresses in the stack area.

SP is assigned to address 106H of SFR. It is incremented before data are moved into stack memory and decremented after data are fetched from stack memory.

After a reset, SP is undetermined, but system programs initialize it to 7FH. After SP is initialized, the application is called.

---

**Caution:** The stack is used from RAM bank 0 address 80H upwards (towards 0ffH). The clock function uses up to 20 bytes of the stack, leaving 108 bytes for the application.  
When the PUSH instruction is executed, data are stored only after SP was incremented.  
Also when PUSH or POP are used during access of RAM bank 1, the data will be stored in the RAM bank 0 stack area.

---

**Table 2.9 Stack pointer (SP)**

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SP	106H	R/W	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
Reset			X	X	X	X	X	X	X	X

When the PUSH instruction is executed, SP is incremented and the SFR and RAM data specified by the operand are moved out. When the POP instruction is executed, the data are moved back into the SFR and RAM specified by the operand, and SP is decremented.

Also when RAM bank 1 was specified for a PUSH or POP operation, data are stacked in RAM bank 0. When the RAM address is used as operand, bank 0 (not bank 1) is accessed.

When a CALL instruction is executed, SP is incremented, and the lower 8 bits of the program counter (PC) are moved to the stack. Then SP is incremented and the upper 8 bits of the PC are moved to the stack. When a RET instruction is executed, data specified by SP are stored as the upper 8 bits of the PC, SP is decremented, and the data specified by the SP are stored as the lower 8 bits of the PC. SP is then decremented further.

When an interrupt is received, SP is incremented, and the lower 8 bits of the PC are moved to the stack. Then SP is incremented again, and the upper 8 bits of the PC are moved to the stack. When a RET1 instruction for returning from interrupt processing is executed, the upper 8 bits of the PC are stored, SP is decremented, and the data specified by the SP are stored as the lower 8 bits of the PC. SP is then decremented further.

## Table Reference Register (TRR)

The table reference register (TRR) is a 16-bit register that serves for ROM and flash memory addressing. The lower byte (TRL) is assigned to address 104H of SFR and the upper byte (TRH) to address 105H of SFR. During reset, the register is initialized to 00H.

The table lookup instruction (LDC) adds the data stored in the TRR to the data stored in the accumulator and uses the result as address for reading data and transferring them to the accumulator. During flash memory read/write (using OS programs), the data stored in the TRR are used as address for the specified bank.

**Table 2.10 Table reference register (lower byte) (TRL)**

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TRL	104H	R/W	TRL7	TRL6	TRL5	TRL4	TRL3	TRL2	TRL1	TRL0
Reset			0	0	0	0	0	0	0	0

**Table 2.11 Table reference register (upper byte) (TRH)**

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TRH	105H	R/W	TRH7	TRH6	TRH5	TRH4	TRH3	TRH2	TRH1	TRH0
Reset			0	0	0	0	0	0	0	0

# CHANGE Instruction

The CHANGE instruction serves for switching between the system BIOS and the application. When a system program is running, the instruction causes a change to the application mode. The program counter is reset to the address specified by label or address.

## Format

CHANGE <label name or address>

## Operation

As described below, operation of the CHANGE instruction differs, depending on whether it is executed while a system program or the application is running. The actual mode shift occurs after the dedicated macro instruction was executed.

### 1) System program running

The system switches from the system program to the application (game mode). The program counter is reset to the application address specified by label or address.

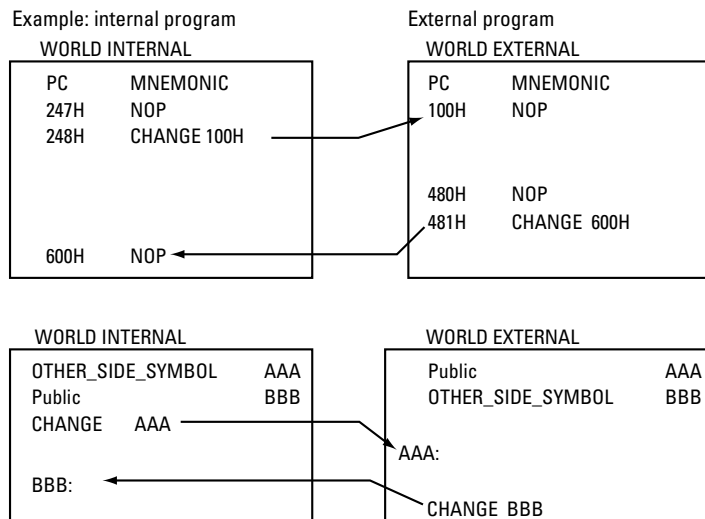
### 2) Application running

The system switches from the application (game mode) to the system program.

However, if bit 1 (LDCEXT) of the external memory control register is set, the CHANGE instruction will not cause a change to the system program. The application continues to run.

The program counter is reset to the system program address specified by label or address.

## Sample program



**Figure 2.21** System Program ↔ Application Transition



# Peripheral System Configuration

This section gives details about peripheral devices including I/O ports, timer, serial communication, etc.

## I/O Ports

The VMU custom chip has three I/O ports which are all mapped to memory using Special Function Registers (SFR). For ports 1 and 3, data direction register (PnDDR) determine the I or O assignment. Port 1 is used only for the serial interface and dedicated Dreamcast interface. Port 7 is a dedicated input port for the VMU buttons.

After a reset, all ports are set as input ports, and the port latch is "0".

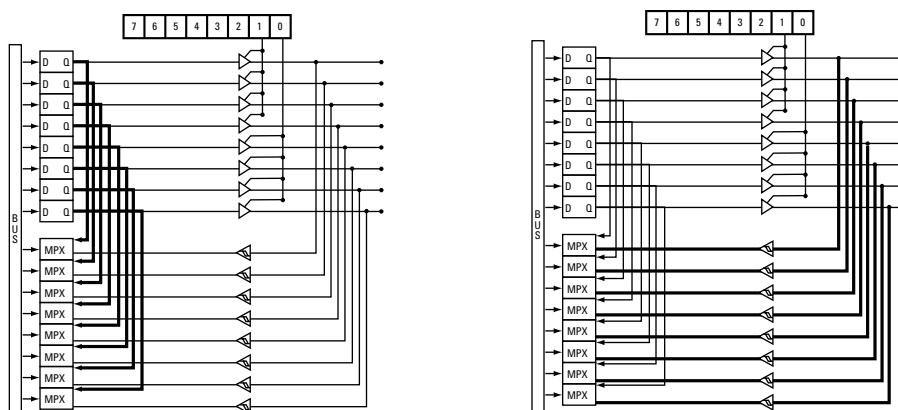
To use the I/O ports, the following Special Function Registers must be operated.

Port 1 (P1)	• P1	• P1DDR	• P1FCR	
Port 3 (P3)	• P3	• P3DDR	• P3INT	• EXT
Port 7 (P7)	• P7	(dedicated input port)		

---

**Caution:** When reading an I/O port, depending on the instruction, data may be either latched (Figure 2.22, "Instruction and Data Path,") or read directly from the port (Figure 2.22, "Instruction and Data Path,"). This must be taken into consideration when reading I/O port data. When reading an I/O port, some instructions read port latched data. BPC, DBNZ, INC, DEC, SET1, CLR1, NOT1

---



**Figure 2.22** Instruction and Data Path

## Port 1

Port 1 can be used as I/O port for the serial interface of the VMU, or for the dedicated Dreamcast interface.

Applications can use only SIO (P10 - P15). To operate these registers, be sure to use bit-level instructions. For details on SIO output, refer to the section on "Serial Interface".

**Caution:** When coding VMU applications, the following operations must be included.

**Standalone operation (SIO not used)**

1. Monitor port 7 to detect 5V.
2. Store values of bits 2 and 5 of port 1.
3. When 5V is detected, change bits 2 and 5 of port 1 to port data output mode and output "0" for these bits.
4. Reset stored values of bits 2 and 5 of port 1.

If these operations are not performed, the VMU may not be recognized correctly when connected to the Dreamcast.

Except for the above operations and for serial data transfer, port 1 registers should not be operated by an application.

**Table 2.12** Port 1 latch (P1): 144H

Port 1	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	P17	P16	P15	P14	P13	P12	P11	P10
Function	Pulse output	TEST	SCK1	SB1	S01	SCK0	SB0	S00

**Table 2.13 Port 1 data direction register (P1DDR): 145H**

Stmbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
P1DDR	145H	W	P17DDR	P16DDR	P15DDR	P14DDR	P13DDR	P12DDR	P11DDR	P10DDR
Reset			0	0	0	0	0	0	0	0

**Caution:** The data direction register for port 1 is a write-only register corresponding to each data latch bit. When a bit operation instruction or an instruction such as INC, DEC, or DBNZ is used for a write-only register, bits other than the specified bit become "1". For the P1DDR, use the following instructions.  
 MOV, MOV @, ST, ST @, POP

Bit name	Function
P17DDR (bit 7)	Input control
P10DDR (bit 7)	0: Input mode 1: Output mode

**P1nDDR (bit 7 to 0): P17 - P10 I/O control**

Specifies whether bits 7 to 0 of port 1 are used for input or output.

When set to "1", P1n is in output mode.

When reset to "0", P1n is in input mode.

**Table 2.14 Port 1 function control register (P1FCR): 146H**

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
P1FCR	146H	W	P17FCR	P16FCR	P15FCR	P14FCR	P13FCR	P12FCR	P11FCR	P10FCR
—			0	0	0	0	0	0	0	0

**Caution:** The data direction register for port 1 is a write-only register. When a bit operation instruction or an instruction such as INC, DEC, or DBNZ is used for a write-only register, bits other than the specified bit become "1". For the P1FCR, use the following instructions.  
 MOV, MOV @, ST, ST @, POP

Bit name	Function
P17CR (bit 7)	P17 control function
	0: Port data (P17) output 1: PWM output
P16CR (bit 6)	Use prohibited
	0: Port data (P16) output (fixed) 1: Not allowed
P15CR (bit 5)	P15 control function
	0: Port data (P15) output 1: Serial interface data (SCK1) output
P14CR (bit 4)	P14 control function
	0: Port data (P14) output 1: Serial interface data (SB1) output
P13CR (bit 3)	P13 control function
	0: Port data (P13) output 1: Serial interface data (SO1) output
P12CR (bit 2)	P12 control function
	0: Port data (P12) output 1: Serial interface data (SCK0) output
P11CR (bit 1)	P11 control function
	0: Port data (P11) output 1: Serial interface data (SB0) output
P10CR (bit 0)	P10 control function
	0: Port data (P10) output 1: Serial interface data (SO0) output

### **P17FCR (bit 7): Select P17 function**

Controls the PWM assigned to P17. When set to "1", the logical sum of the PWM signal and the port latch data is output. When reset to "0", the port latch data are output.

### **P16FCR (bit 6): Select P16 function**

This bit is fixed to "0". It may not be manipulated by an application.

### **P15FCR (bit 5): Select P15 function**

Controls the clock assigned to P15 for serial transfer 1. When set to "0", the logical sum of the serial interface clock (SCK1) and port latch data is output. When reset to "0", port latch data are output.

### **P14FCR (bit 4): Select P14 function**

Controls the data assigned to P14 for serial transfer 1. When set to "0", the logical sum of the serial interface data (SB1) and port latch data is output. When reset to "0", port latch data are output.

### **P13FCR (bit 3): Select P13 function**

Controls the data assigned to P13 for serial transfer 1. When set to "0", the logical sum of the serial interface data (S01) and port latch data is output. When reset to "0", port latch data are output.

### **P12FCR (bit 2): Select P12 function**

Controls the clock assigned to P12 for serial transfer 0. When set to "0", the logical sum of the serial interface clock (SCK0) and port latch data is output. When reset to "0", port latch data are output.

### **P11FCR (bit 1): Select P11 function**

Controls the data assigned to P11 for serial transfer 0. When set to "0", the logical sum of the serial interface data (SB0) and port latch data is output. Serial interface data can always be input.

### **P10FCR (bit 0): Select P10 function**

Controls the data assigned to P10 for serial transfer 0. When set to "0", the logical sum of the serial interface data (S00) and port latch data is output. When reset to "0", port latch data are output.

- 
- Caution:**
- To use the function assigned to port 1, the corresponding port latch must be reset to "0". For example, to use PWM, set P17FCR to "0" and reset P17 to "0".
  - The instructions BPC, DBNZ, INC, DEC, SET1, CLR1, NOT1 read port latch data. Other instructions read data assigned to the port.
-

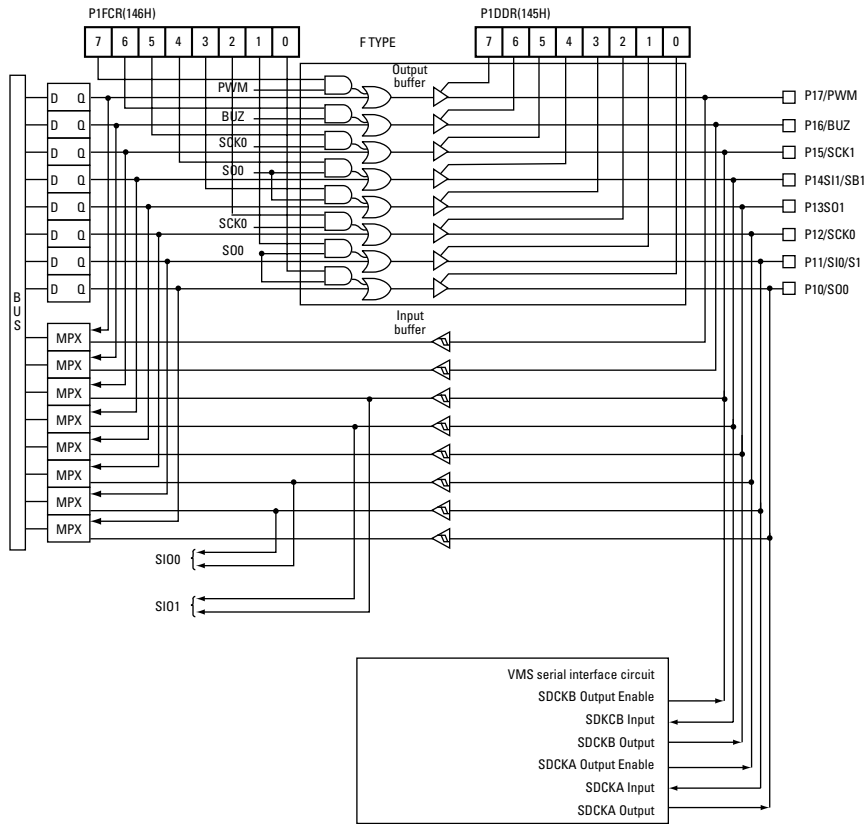


Figure 2.23 Port 1 Block Diagram

## Port 3

Port 3 is an input-only port dedicated to the VMU direction buttons, A button, B button, MODE button, and SLEEP button.

Table 2.15 Port 3 latch (P3)

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
P3	14CH	R/W	P37	P36	P35	P34	P33	P32	P31	P30
Function	SLEEP	MODE	B button	A button	RIGHT	LEFT	DOWN	UP		
Reset			0	0	0	0	0	0	0	0

Bits 0 to 7 of port are programmable pull-up bits. The application must set the bit corresponding to the button to be detected to "0". When the button is pressed, the bit is reset to "0".

**Caution:** Measures against simultaneous presses of different direction buttons must be taken by the application.

**Port 3 data direction register (P3DDR): 14DH**

This register may not be manipulated by an application.

**Table 2.16 Port 3 interrupt control register (P3INT)**

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
P3INT	14EH	R/W	-	-	-	-	-	P32INT	P31INT	P30INT
Reset			H	H	H	H	H	0	0	0

Bit name	Function
P32INT (bit 2)	Port 3 interrupt control flag
	0: Port interrupt disabled 1: Port interrupt enabled
P31INT (bit 1)	Port 3 interrupt source flag
	0: Interrupt source disabled 1: Interrupt source enabled
P30INT (bit 0)	Port 3 interrupt request enable
	0: Interrupt request disabled 1: Interrupt request enabled

**P32INT (bit 2): port 3 interrupt generation selector flag**

Determines whether an interrupt is generated while a button connected to port 3 is pressed. Whereas the P30INT (bit 0) flag selects whether a generated interrupt is accepted or not, this flag controls interrupt generation itself.

When reset to "0", no interrupt is generated.

When set to "1", an interrupt is generated.

**Caution:** The port 3 interrupt is a level interrupt which is generated continuously for as long as the button is pressed.

**P31INT (bit 1): port 3 interrupt source flag**

This flag is relevant if the P32INT flag is set. The port 3 interrupt request status is monitored, and the flag is set to "1" when an interrupt request is generated by port 3. When no interrupt request is generated, the flag does not change. This allows an interrupt processing routine to specify an interrupt source.

**Caution:** This flag must be reset by the application. Use a suitable interrupt processing routine to do this.

## P30INT (bit 0): port 3 interrupt request enable control

Enables (1) or disables (0) interrupt requests from port 3. When reset to "0", interrupt processing is disabled and the interrupt processing routine is not called. When set to "1", the interrupt vector 004BH is called when an interrupt is generated (P31INT = 1).

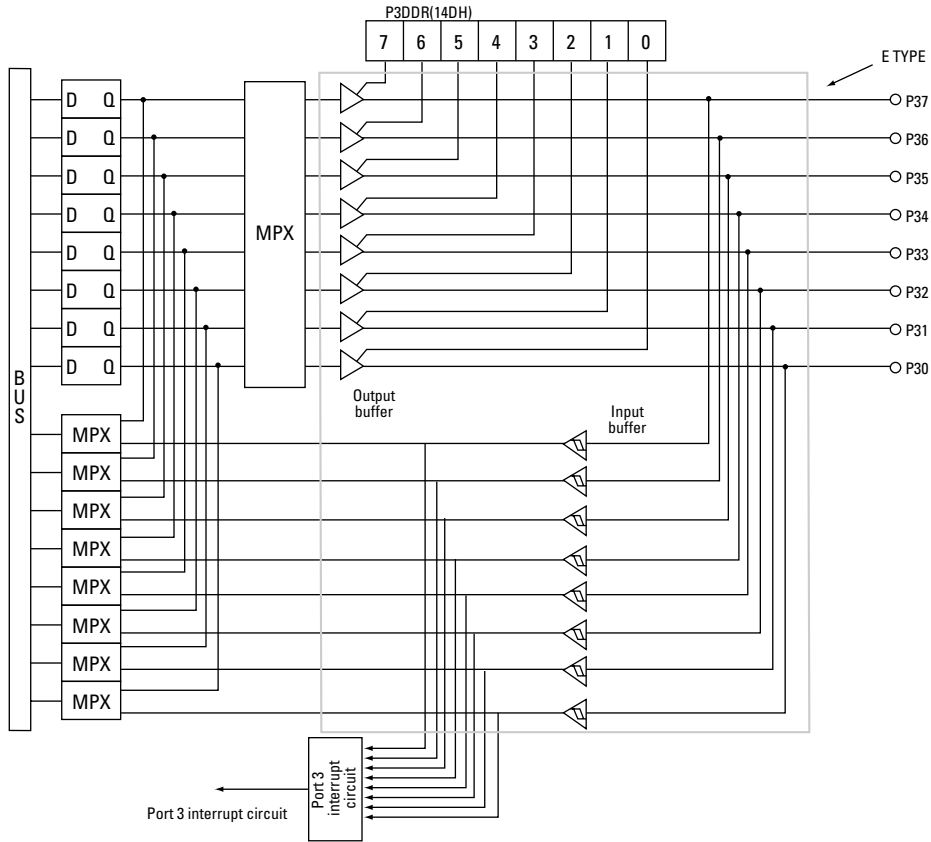


Figure 2.24 Port 3 block diagram

## Port 7

Port 7 is a dedicated input port that serves for low-voltage detection and for checking the connection status to the Dreamcast.

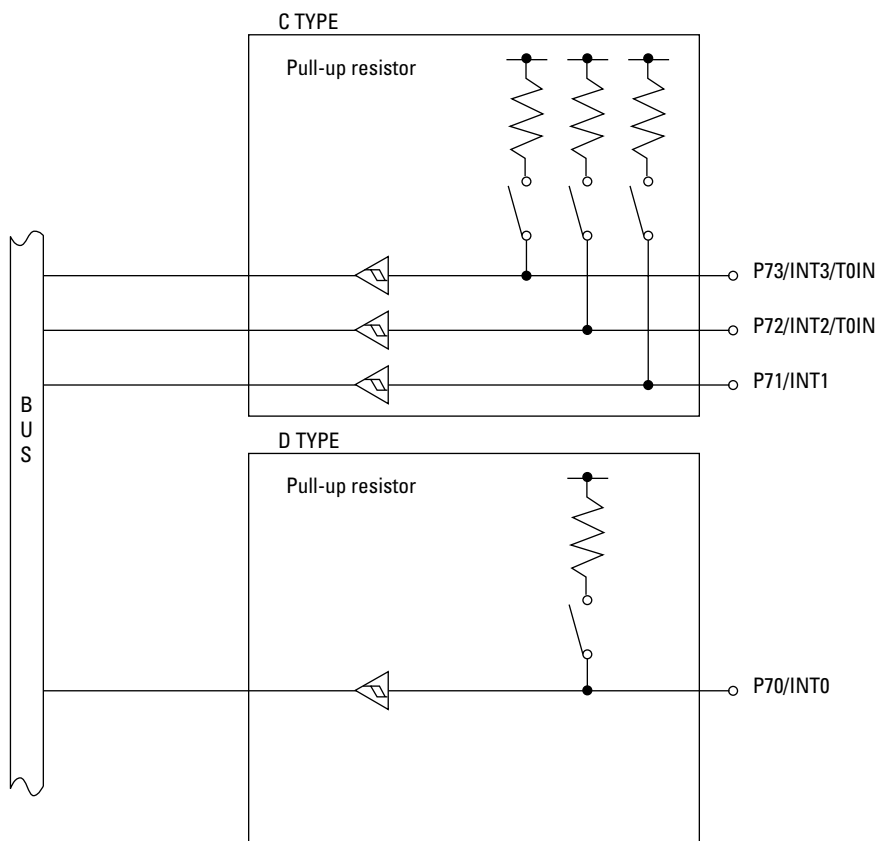


**Table 2.17 Port 7 (P7)**

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
P7	15CH	R	-	-	-	-	P73	P72	P71	P70
Function			-	-	-	-	ID1	ID0	Low voltage	5V detection
Reset			H	H	H	H	0	0	1	0

Bits 0 to 3 of port 7 are pulled up. Immediately after a reset, bit 1 is set to "0" and all other bits are reset to "0".

**Caution:** VMU applications should be designed in such a way as to store data in flash memory and then terminate quickly when the VMU is connected to the Dreamcast controller while the application is running.  
The application should monitor bit 0 of port 7. When detecting that bit was set to "0", the same application termination routine as when the MODE button is pressed should be carried out, and control should be returned to the system BIOS. The interrupt INT0 can be used to detect the connection status.



**Figure 2.25** Port 7 Block Diagram

# Timer/Counter 0 (T0)

The timer/counter 0 (T0) in the VMU custom chip is a 16-bit timer/counter with the following 4 functions. The prescaler of timer 0 is an 8-bit type.

The following Special Function Registers are used to control the timer/counter 0.

T0H, T0HR, T0L, T0LR, T0CNT, T0PRR, ISL, I23CR

- Mode 0: 8-bit reload timer x 2 channels
- Mode 1: 8-bit reload timer + 8-bit reload counter
- Mode 2: 16-bit reload timer
- Mode 3: 16-bit reload counter

## Functions

### 8-bit reload timer x 2 channels (mode 0)

The clock from the 8-bit prescaler is used to drive two separate 8-bit reload timers (T0H, T0L).

### 8-bit reload timer + 8-bit reload counter (mode 1)

T0H operates as an 8-bit reload timer driven by the prescaler clock. T0L performs counting by detecting the input signal at the P72/INT2/T0IN and P73/INT3/T0IN pins.

### 16-bit reload timer (mode 2)

The clock from the 8-bit prescaler is used to drive the 16-bit reload timer (T0H + T0L).

### 16-bit reload counter (mode 3)

The overflow of T0L is used as clock for T0H, to drive the 16-bit reload counter. T0L counts the input signal at the P72/INT2/T0IN and P73/INT3/T0IN pins.

### Interrupt generation

When the interrupt enable bit is set, overflow of the register T0H or T0L generates a T0H or T0L interrupt.

## Circuit Configuration

The timer/counter 0 (T0) configuration is shown in below.

### Prescaler ... ②

The prescaler is an 8-bit programmable counter that operates constantly while the system is on. The cycle clock is a signal generated with each cycle when an instruction is executed and at HALT mode.

### Timer/counter 0 low (T0L) ... →

This 8-bit reload timer/counter uses the prescaler output or an external signal (from other VMU) as a clock. In modes 0 and 1, T0L is the overflow. In modes 2 and 3, T0H is the overflow. The T0LR (reload register) contents are reloaded to the respective counter. Reloading is carried out also when the T0LRUN (T0CNT bit 6) is reset and the counter stops.

### Timer/counter 0 high (T0H) ... ↗

This 8-bit reload timer/counter uses the prescaler output or the T0L overflow as a clock. At T0H overflow the contents of the timer 0 high reload register (T0HR) are reloaded. Reloading is carried out also when the T0HRUN (T0CNT bit 7) is reset and the counter stops.

### Timer/counter 0 control register (T0CNT) ... ③

Serves for T0 mode 0 to 3 setting and interrupt control.

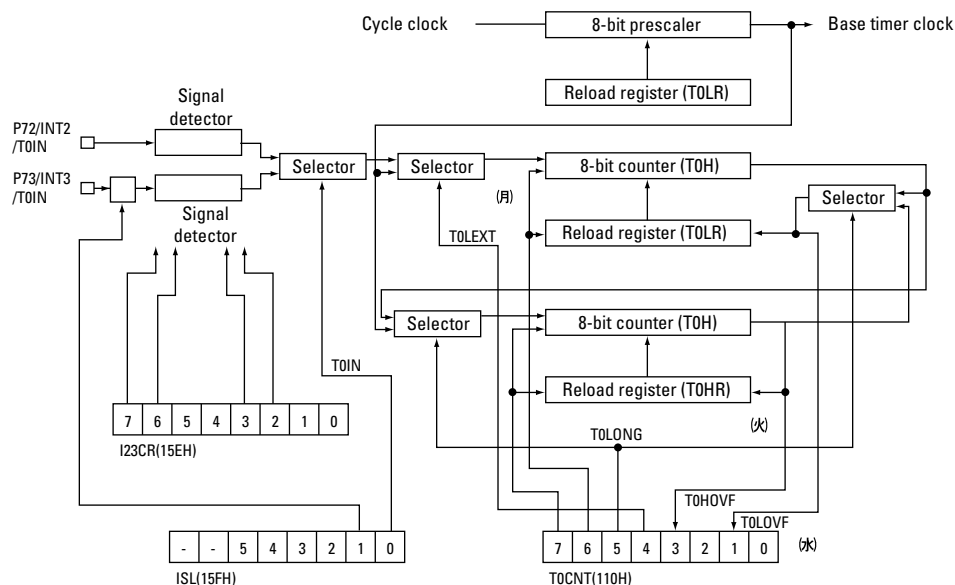


Figure 2.26 Timer/Counter 0 Block Diagram

## Related Registers

**Table 2.18** *Timer/counter 0 control register (TOCNT)*

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TOCNT	14EH	R/W	POHRUN	POLRUN	POLONG	POLEXT	POHOVF	TOHIE	TLOVVF	TOLIE
Reset			0	0	0	0	0	0	0	0

Bit name	Function
POHRUN (bit 7)	TOH count control
	0: Count stop/data reload 1: Count start
POLRUN (bit 6)	TOL count control
	0: Count stop/data reload 1: Count start
POLONG (bit 5)	Timer/counter 0 bit length selector
	0: 8 bit 1: 16 bit
POLEXT (bit 4)	TOL input clock select
	0: Prescaler output 1: External pin input signal Pin for external input can be specified by input select register (ISL)
POHOVF (bit 3)	TOH overflow flag
	0: No overflow flag 1: Overflow flag
TOHIE (bit 2)	TOH interrupt request enabled
	0: Interrupt request disabled 1: Interrupt request enabled
TLOVVF (bit 1)	TOL overflow flag
	0: No overflow flag 1: Overflow flag
TOLIE (bit 0)	TOL interrupt request enabled
	0: Interrupt request disabled 1: Interrupt request enabled

**T0HRUN (bit 7): T0H count control**

Controls count-up start/stop of timer/counter 0 high (T0H). When set to "0", the T0H clock is supplied and counting starts. To stop counting, the bit must be reset to "0". This stops the clock and sends the reload data (T0HR) to T0H.

**T0LRUN (bit 6): T0L count control**

Controls count-up start/stop of timer/counter 0 low (T0L). When set to "0", the T0L clock is supplied and counting starts. To stop counting, the bit must be reset to "0". This stops the clock and sends the reload data (T0LR) to T0L.

**T0LONG (bit 5): Timer/counter 0 bit length select**

Specifies the bit length of T0. "0" selects a 16-bit counter and "0" an 8-bit counter.

**Table 2.19 For mode 0 or 1, specify "0", and for mode 2 or 3, specify "0".**

Mode	T0LONG	T0EXT
0	0	0
1	0	1
2	1	0
3	1	1

**T0EXT (bit 4): T0L input clock select**

Specifies the clock supplied to T0L. The clock can be either an external signal (from a another connected VMU) or the prescaler output.

When the bit is set to "0", the external input signal is selected. When the bit is reset to "0", the prescaler output is selected.

When the external signal is selected (1), either port P72 (INT2/T0IN pin) or P73 (INT3/T0IN pin) can be used. as T0L clock. Switching between P72 and P73 is performed by the input signal select register (ISL).

**T0HOVF (bit 3): T0H overflow flag**

This flag is set when T0H overflow has occurred. If there is no overflow, the flag does not change.

This flag must be reset by the T0H interrupt processing routine or another routine of the application.

**T0HIE (bit 2): T0H interrupt request enable control**

Enables or disables interrupt request generation at T0H overflow.

When set to "0", the interrupt vector 0023H is called when T0H overflow occurs. When reset to "0", no interrupt request is generated.

### T0LOVF (bit 1): T0L overflow flag

This flag is set when T0L overflow has occurred. If there is no overflow, the flag does not change.

This flag must be reset by the T0L interrupt processing routine or another routine of the application.

When the 16-bit counter is used, the flag is not set also when overflow occurs. When T0H overflow occurs, it is set together with T0HOVF.

### T0LIE (bit 0): T0L interrupt request enable control

Enables or disables interrupt request generation at T0L overflow.

When set to "0", the interrupt vector 0013H is called when T0H overflow occurs. When reset to "0", no interrupt request is generated.

- 
- Caution:**
- The overflow flags (T0HOVF, T0LOVF) must be reset to "0" by the respective interrupt processing routine of the application.
  - When using the 16-bit counter, set T0HRUN and T0LRUN together to "0".
  - When using the 16-bit counter, set T0HOVF and T0LOVF together to "0".
- 

### Input signal select register (ISL)

This register serves to select the time constant for the noise filter connected to P73 (INT3/T0IN pin) and to select the external signal.

- 
- Caution:** This register may not be manipulated by an application.
- 

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ISL	15FH	R/W	-	-	ISL5	ISL4	ISL3	ISL2	ISL1	ISL0
Reset			H	H	0	0	0	0	0	0

Bit name	Function		
ISL5 (bit 5) ISL4 (bit 4)	Base timer clock select		
	ISL5	ISL4	
	1 0 X	1 1 0	Timer/counter T0 prescaler Cycle clock Quartz oscillator
ISL3 (bit 3)	Use prohibited		
	0: fBST/16 (fixed) 1: Not allowed		
ISL2 (bit 2) ISL1 (bit 1)	Noise filter time constant select		
	ISL2	ISL1	
	1 0 X	1 1 X	16Tcyc 64Tcyc 1Tcyc
ISL0 (bit 0)	T0 clock input pin select		
	0: P72/INT2/TOIN pin 1: P73/INT3/TOIN pin		

**ISL5, ISL4 (bits 5, 4): base timer clock select**

Select the base timer input clock.

ISL5	ISL4	Base timer input clock
0	0	Quartz oscillator

**ISL3 (bit 3): prohibited**

Use of this bit is prohibited.

**ISL2, ISL1 (bits 2, 1): noise filter time constant select**

Selects the time constant of the noise filter.

ISL2	ISL1	Time constant
0	0	1Tcyc

The table below shows values for the signal time constant and noise range figures.

Time constant	Noise *1	Noise/
signal *2	Noise *3	
1Tcyc	< 1Tcyc	1Tcyc –
2Tcyc	2Tcyc <	

- Caution:**
- A signal not matching the time constant conditions is considered noise and is not input.
  - Sometimes even a signal matching the time constant conditions may be considered noise and not input.
  - A signal matching the time constant conditions is considered normal and is input.

### ISL0 (bit 0): T0 clock input select

Sets port T0 external signal input to P73 (INT3/T0IN pin) or P72 (INT2/T0IN pin).

When reset to "0", the signal at P72 (INT2/T0IN pin) is used as T0 clock.

When set to "0", the signal at P73 (INT3/T0IN pin) is used as T0 clock.

### Timer 0 prescaler register (TOPRR)

The timer 0 prescaler register (TOPRR) serves for setting the timer/counter 0 clock frequency. The 8-bit programmable counter allows 256 different settings.

The 8-bit prescaler uses the cycle clock directly as its clock. By setting the desired data in TOPRR (111H), the timer/counter 0 clock frequency TPR can be set.

8-bit prescaler:  $TPR = 1 \times (256 - [TOPRR])$  (decimal)

Tcyc: Cycle clock

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TOPRR	111H	R/W	TOPRR7	TOPRR6	TOPRR5	TOPRR4	TOPRR3	TOPRR2	TOPRR1	TOPRR0
Reset			0	0	0	0	0	0	0	0

### Timer 0 low register (TOL)

This is an 8-bit timer/counter.

It selects whether the output of the prescaler or the external signal from P72 (INT2/T0IN pin) or P73 (INT3/T0IN pin) is used as clock signal.



The clock is used for count-up. When overflow occurs, the overflow flag is set and an interrupt is generated.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TOL	112H	R	TOL7	TOL6	TOL5	TOL4	TOL3	TOL2	TOL1	TOL0
Reset			0	0	0	0	0	0	0	0

#### Timer 0 low reload register (TOLR)

The data for reloading in the timer/counter 0 low (TOL) are set in this register. When using 8-bit mode, the contents of this register are reloaded into TOL.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TOLR	113H	R/W	TOLR7	TOLR6	TOLR5	TOLR4	TOLR3	TOLR2	TOLR1	TOLR0
Reset			0	0	0	0	0	0	0	0

#### Timer 0 high register (TOH)

This is an 8-bit timer/counter.

Count-up is performed with the prescaler output or the TOL overflow (T0HOVF). When overflow occurs, the overflow flag is set and an interrupt is generated.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TOH	114H	R	TOH7	TOH6	TOH5	TOH4	TOH3	TOH2	TOH1	TOH0
Reset			0	0	0	0	0	0	0	0

#### Timer 0 high reload register (TOHR)

The data for reloading in the timer/counter 0 high (TOH) are set in this register. When TOH overflow has occurred and when the count was stopped (TOHRUN = 0), the contents of this register are reloaded into TOH.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TOHR	115H	R/W	TOHR7	TOHR6	TOHR5	TOHR4	TOHR3	TOHR2	TOHR1	TOHR0
Reset			0	0	0	0	0	0	0	0

## External interrupt 2, 3 control register (I23CR)

Sets external signal detection and interrupt.

ISL0	I23CR7	I23CR6	I23CR3	I23CR2	External signal condition
1	0	1	-	-	P73/INT3/TOIN falling edge count
1	1	0	-	-	P73/INT3/TOIN rising edge count
1	1	1	-	-	P73/INT3/TOIN dual edge count
0	-	-	0	1	P72/INT2/TOIN falling edge count
0	-	-	1	0	P72/INT2/TOIN rising edge count
0	-	-	1	1	P72/INT2/TOIN dual edge count
-	0	0	0	0	No count

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I23CR	15EH	R/W	I23CR7	I23CR6	I23CR5	I23CR4	I23CR3	I23CR2	I23CR1	I23CR0
Reset			0	0	0	0	0	0	0	0

In combination with the input signal select register ISL0, this register specifies the count conditions. Possible combinations are shown in the following table.

Bit	Function
I23CR7 (bit 7)	INT3 rising edge detection control
	0: No detect 1: detect
I23CR6 (bit 6)	INT3 falling edge detection control
	0: No detect 1: detect
I23CR5 (bit 5)	INT3 interrupt source
	0: Interrupt source disabled 1: Interrupt source enabled
I23CR4 (bit 4)	INT3 interrupt request enabled
	0: Interrupt request disabled 1: Interrupt request enabled
I23CR3 (bit 3)	INT2 rising edge detection control
	0: No detect 1: detect

I23CR2 (bit 2)	INT3 falling edge detection control
	0: No detect 1: detect
I23CR1 (bit 1)	INT2 interrupt source
	0: Interrupt source disabled 1: Interrupt source enabled
I23CR0 (bit 0)	INT2 interrupt request enabled
	0: Interrupt request disabled 1: Interrupt request enabled

**I23CR7 (bit 7): INT3 rising edge detection control**

Specifies whether to detect the rising edge of the interrupt signal at P73 (INT3/T0IN pin).

When set to "0", the rising edge of the INT3 signal at P73 is detected. When the INT3 interrupt is generated, I23CR5 is set to "0" and the interrupt processing routine specified by the interrupt vector is called, if interrupt request is enabled (I23CR4 = 1).

When reset to "0", the rising edge of the interrupt signal is not detected.

**I23CR6 (bit 6): INT3 falling edge detection control**

Specifies whether to detect the falling edge of the interrupt signal at P73 (INT3/T0IN pin).

When set to "0", the falling edge of the INT3 signal at P73 is detected. When the INT3 interrupt is generated, I23CR5 is set to "0" and the interrupt processing routine specified by the interrupt vector is called, if interrupt request is enabled (I23CR4 = 1).

When reset to "0", the rising edge of the interrupt signal is not detected.

**I23CR5 (bit 5): INT3 interrupt source**

When interrupt edge detection at P73 (INT3/T0IN pin) has occurred, this flag is set.

The flag must be reset by the interrupt processing routine of the application.

**I23CR4 (bit 4): INT3 interrupt request enable**

Enables or disables the INT3 interrupt.

If set to "0", the INT3 interrupt vector is called when the I23CR5 flag is set.

If reset to "0", the interrupt processing routine is not called, also when an interrupt is generated.

### **I23CR3 (bit 3): INT2 rising edge detection control**

Specifies whether to detect the rising edge of the interrupt signal at P72 (INT2/T0IN pin).

When set to "0", the rising edge of the INT2 signal at P72 is detected. When the INT2 interrupt is generated, I23CR1 is set to "0" and the interrupt processing routine is called, if interrupt request is enabled (I23CR4 = 0).

### **I23CR2 (bit 2): INT2 falling edge detection control**

Specifies whether to detect the falling edge of the interrupt signal at P72 (INT2/T0IN pin).

When set to "0", the falling edge of the INT2 signal at P72 is detected. When the INT2 interrupt is generated, I23CR1 is set to "0" and the interrupt processing routine is called, if interrupt request is enabled (I23CR4 = 0).

### **I23CR1 (bit 1): INT2 interrupt source**

When interrupt edge detection at P72 (INT2/T0IN pin) has occurred, this flag is set.

The flag must be reset by the interrupt processing routine of the application.

### **I23CR0 (bit 0): INT2 interrupt request enable**

Enables or disables the INT3 interrupt.

If set to "0", the INT3 interrupt vector is called when the I23CR1 flag is set.

If reset to "0", the interrupt processing routine is not called, also when an interrupt is generated.

- 
- Caution:**
- When I23CR7 and I23CR6, or I23CR3 and I23CR2 are both "0", edge detection is not performed. When both are "0", both edges are detected.
  - Input from P73 (INT3/T0IN pin) is routed through a noise filter.
-

## Circuit Configuration and Operation Principles

### Timer 0 mode setting

Mode	TOLONG	TOLEXT
0	0	0
1	0	1
2	1	0
3	1	1

### Mode 0: 8-bit reload timer x 2 channels

In mode 0, timer 0 functions as an 8-bit reload timer with two channels. The relationship between the timer value and the reload register (TOLR) setting value is as shown below.

Time until T0HOVF is set (1) (decimal) = (256 - T0HR setting value) x TPR

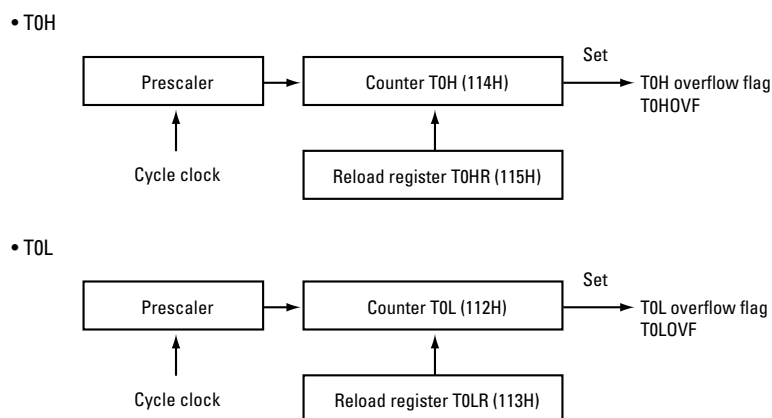
Time until T0LOVF is set (1) (decimal) = (256 - T0LR setting value) x TPR

TPR: Prescaler clock cycle

When the count control bit (T0HRUN, T0LRUN) is set, counting starts. When it is reset, counting stops, and the contents of the reload register (T0HR, T0LR) are sent to the counter (T0H, T0L).

When the timer/counter 0 (T0H, T0L) overflows, the overflow flag (T0HOVF, T0LOVF) is set, and the contents of the reload register (T0HR, T0LR) are sent to the counter (T0H, T0L).

When both the overflow flag (T0HOVF, T0LOVF) and interrupt request enable flag (T0HIE, T0LIE) are set, the interrupt request is signalled to the interrupt control circuit.



**Figure 2.27** Mode 0: 8-Bit Reload Timer x 2 Channel Circuit Configuration

## Mode 0 sample program

• Mode 0 sample program

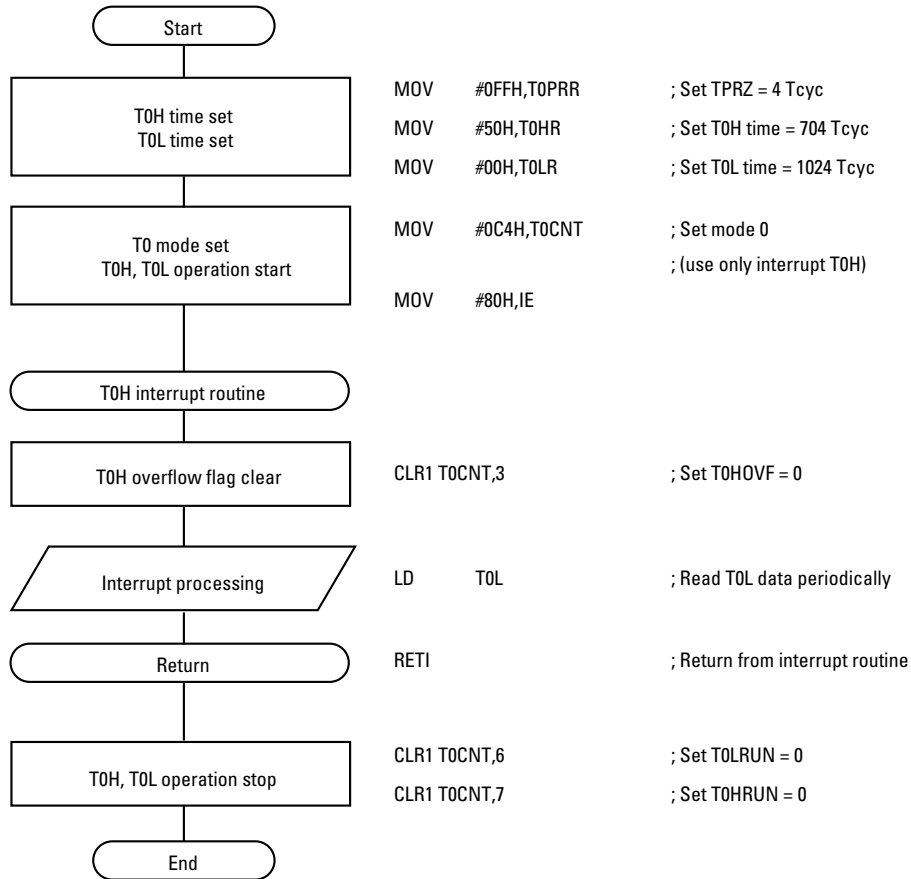


Figure 2.28 Flow Chart and Program

## Mode 1: 8-bit reload timer + 8-bit reload counter

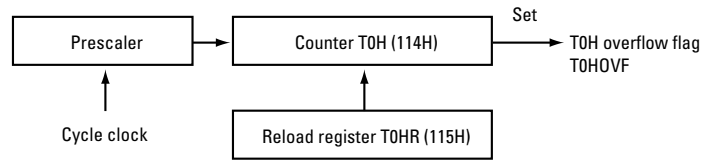
### 8-bit reload timer

The upper 8 bits of timer 0 (TOH) operate as an 8-bit reload timer. The relationship between the timer value and the reload register (TOHR) setting value is as shown below.

$$\text{Time until TOHOVF is set (1) (decimal)} = (256 - \text{TOHR setting value}) \times \text{TPR}$$

TPR: Prescaler clock cycle

Each time TOHOVF is set, the reload register value is sent to the counter TOH. Timer operation continues until the TOH count control bit (TOHRUN) is reset. Operation principles are the same as for mode 0.



**Figure 2.29** Mode 1: 8-Bit Reload Timer (TOH) Block Diagram

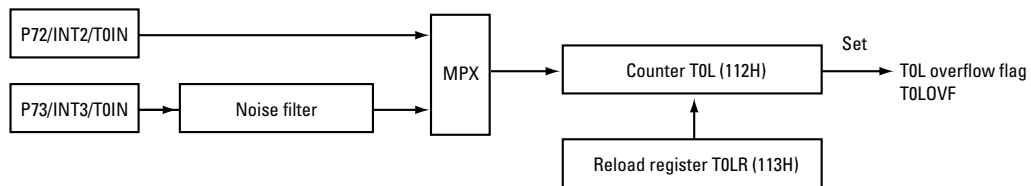
**8-bit reload counter**

The lower 8 bits of timer 0 (TOL) are used to count up the external input signal. This signal is filtered by a noise filter circuit. For details, refer to the section “Input Signal Select Register (ISL)” in “Timer / Counter 0 (T0)”.

The relationship between the count value and the reload register (TOLR) setting value is as shown below.

$\text{Time until TOLOVF is set (1) (decimal)} = 256 - (\text{TOLR setting value})$
---

When the TOL overflow flag (TOLOVF) flag is set, the reload register (TOLR) value is sent to the counter (TOL). Timer operation continues until the TOH count control bit (TOLRUN) is reset.



**Figure 2.30** Mode 1: 8-Bit Reload Timer (TOL) Block Diagram

## Mode 1 sample program

• Mode 1 sample program

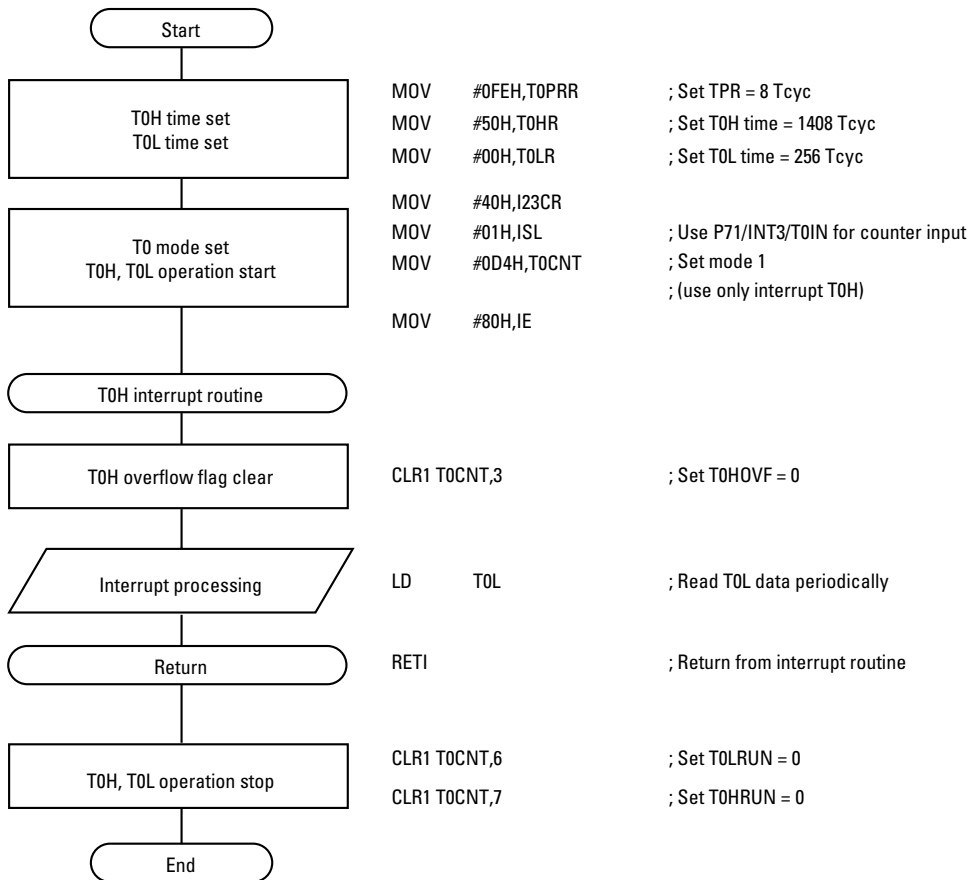


Figure 2.31 Flow Chart and Program

## Mode 2: 16-bit reload timer

In mode 2, T0H and T0L are connected in series and operate as a 16-bit timer.

To start the timer, the count control bits (TOHRUN, TOLRUN) of T0H and T0L must be set simultaneously.

The relationship between the timer value and the reload register (T0HR, T0LR) setting values is as shown below.

$$\begin{aligned} &\text{Time until T0HOVF is set (1) (decimal)} \\ &= (65536 - 256 \times (\text{T0HR setting value}) - (\text{T0LR setting value})) \times \text{TPR} \end{aligned}$$

TPR: Prescaler clock cycle

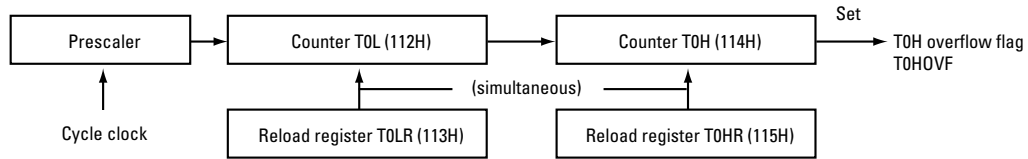
When T0LOVF and T0HOVF are both set, the reload register (T0HR, T0LR) values are sent simultaneously to T0L and T0H when T0HOVF occurs. Timer operation continues until the T0H count control bit is reset. Operation principles are the same as for mode 0.



For reading data from timer 0 (T0), use the following procedure.

```

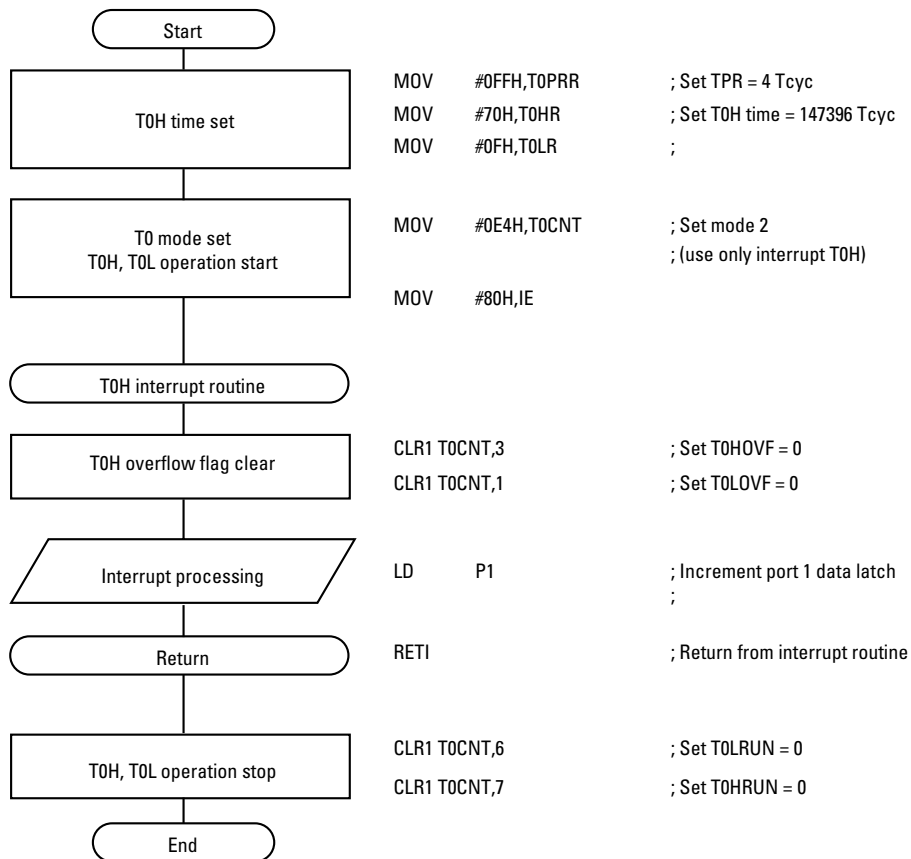
T0L      LD      T0L      ; Read T0L data (1)
↓
        ST      020H
T0H      LD      T0H      ; Read T0H data
↓
        ST      021H
T0L      LD      T0L      ; Read T0L (2) data again
↓
        BP      T0L,7,DES ; When T0L (2) bit 7 is "0"
        BN      020H,7,DES ; and T0L (1) bit 7 is "0"
        ST      020H
T0H      LD      T0H      ; Read T0H (2)
        ST      021H
DES:     -- next program
    
```



**Figure 2.32** Mode 2: 16-Bit Reload Timer Block Diagram

**Mode 2 sample program**

• Mode 2 sample program



**Figure 2.33** Flow Chart and Program

## Mode 3: 16-bit reload counter

In mode 3, T0H and T0L are connected in a cascaded configuration and operate as a 16-bit counter. The signal input to P72 (INT2/T0IN pin) or P73 (INT3/T0IN pin) is used as clock signal. Input signal selection is carried out by the ISL register of SFR. The input from P73 (INT3/T0IN pin) is routed through a noise filter.

To start the timer, the count control bits (T0HRUN, T0LRUN) of T0H and T0L must be set simultaneously.

The relationship between the count value and the reload register (T0HR, T0LR) setting values is as shown below.

$\text{Time until T0HOVF is set (1) (decimal)} = 65536 - 256 \times (\text{T0HR setting value}) - (\text{T0LR setting value})$
--

When T0LOVF and T0HOVF are both set, the reload data (T0HR, T0LR) are sent simultaneously to T0L and T0H when T0HOVF occurs. Timer operation continues until the count control bit is reset. Operation principles are the same as for mode 0.

For reading data from timer 0 (T0), use the following procedure.

```

T0L          LD          T0L          ; Read T0L data (1)
↓
              ST          020H
T0H          LD          T0H          ; Read T0H data
↓
              ST          021H
T0L          LD          T0L          ; Read T0L (2) data again
↓
              BP          T0L,7,DES   ; When T0L (2) bit 7 is "0"
              BN          020H,7,DES  ; and T0L (1) bit 7 is "0"
              ST          020H
T0H          LD          T0H          ; Read T0H (2)
              ST          021H
DES:         -- next program
    
```

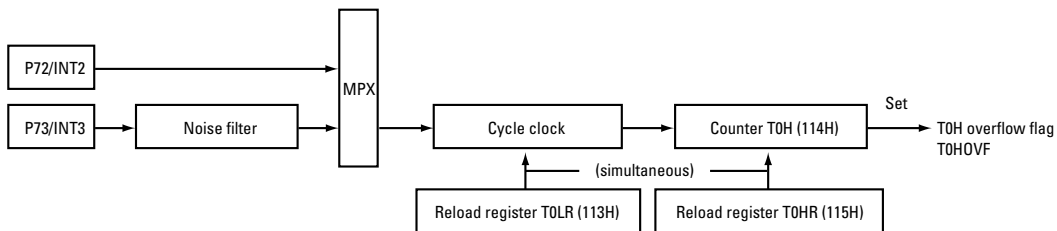
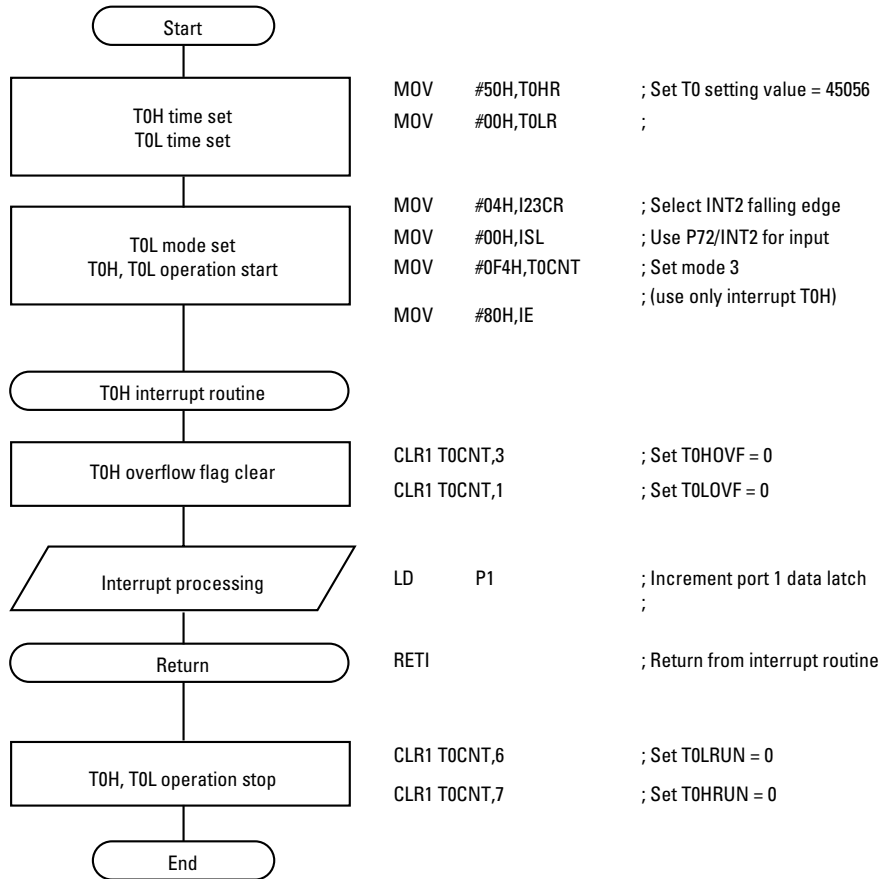


Figure 2.34 Mode 3: 16-Bit Reload Timer Block Diagram

**Mode 2 sample program**

• Mode 3 sample program



**Figure 2.35** Flow Chart and Program

# Timer 1 (T1)

The timer/counter 1 (T1) in the VMU custom chip is a 16-bit timer with the following 4 functions.

- Mode 0: 8-bit reload timer x 2 channels
- Mode 1: 8-bit reload timer + 8-bit pulse generator
- Mode 2: 16-bit reload timer
- Mode 3: Variable bit length pulse generator (9 to 16 bits)

## Functions

### 8-bit reload timer x 2 channels (mode 0)

The cycle clock is used to drive two separate 8-bit reload timers (T1H, T1L).

### 8-bit reload timer + 8-bit pulse generator (mode 1)

T1H operates as an 8-bit reload timer driven by the cycle clock. T1L operates as an 8-bit pulse generator whose output appears at the P17/pulse output pin.

### 16-bit reload timer (mode 2)

The overflow of T1L is used as clock for T1H, to drive the 16-bit reload timer. The input clock to T1L is the cycle clock. Each time a T1L overflow occurs, the T1LR and T1HR reload data are loaded into T1L and T1H. The T1L clock can be the cycle clock or the cycle clock divided by 2.

### Variable bit length pulse generator (9 to 16 bits) (mode 3)

T1L and T1H can be used to generate a 9 to 16 bit pulse signal. This signal is output via the P17/pulse output pin.

The T1L clock can be the cycle clock or the cycle clock divided by 2.

### Interrupt generation

When the interrupt request enable bit is set, overflow of the register T1H or T1L generates a T1H or T1L interrupt request.

The following Special Function Registers must be operated to control the timer 1 (T1).

T1H, T1HR, T1HC, T1L, T1LR, T1LC, T1CNT, P1

## Circuit Configuration

The timer/counter 1 (T1) configuration is shown in below.

### Timer 1 low (T1L)... ②

This 8-bit reload timer uses the cycle clock or cycle clock divided by 2. When T1L overflow occurs, the T1LR value is reloaded. The T1LR value is reloaded when T1L overflow occurs. Resetting the T1LRUN (T1CNT bit 6) to '0' stops the timer and causes the T1LR data to be transferred to T1L.

### Timer 1 low comparator (T1LC)... →

This circuit consists of the 8-bit timer 1 low comparator data register (T1LC) and an 8-bit data comparator. It compares the data for T1L and T1LC.

### Timer 1 high (T1H)... ➔

This 8-bit reload timer uses the cycle clock or the T1L overflow as clock. At T1H overflow, the T1HR value is reloaded. Reload also occurs when T1HRUN (T1CNT bit 7) is reset to stop the timer.

### Timer 1 high comparator (T1HC)... ③

This circuit consists of the 8-bit timer 1 high comparator data register (T1HC) and an 8-bit data comparator. It compares the data for T1H and T1HC.

### Timer 1 control register (T1CNT)... (

Serves for T1 mode setting and interrupt control.

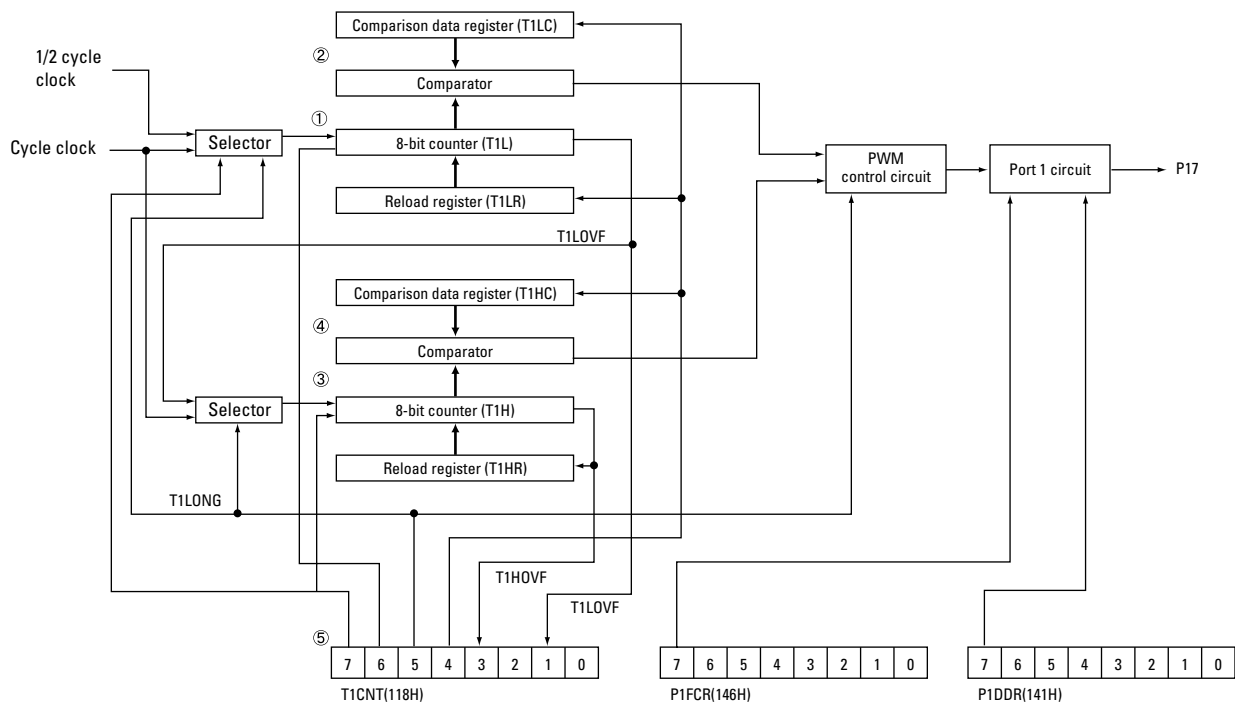


Figure 2.36 Timer 1 Block Diagram

## Related Registers

### Timer 1 control register (T1CNT)

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
T1CNT	118H	R/W	T1HRUN	T1LRUN	T1LONG	ELDT1C	T1HOVF	T1HIE	T1LOVF	T1LIE
Reset			0	0	0	0	0	0	0	0

Bit name	Function
T1HRUN (bit 7)	T1H count control
	0: Count stop/data reload 1: Count start
T1LRUN (bit 6)	T1L count control
	0: Count stop/data reload 1: Count start
T1LONG (bit 5)	Timer 1 bit length selector
	0: 8 bit 1: 16 bit
ELDT1C (bit 4)	Pulse generator data update enabled
	0: Disabled 1: Enabled
T1HOVF (bit 3)	T1H overflow flag
	0: No overflow flag 1: Overflow flag
T1HIE (bit 2)	T1H interrupt request enabled
	0: Interrupt request disabled 1: Interrupt request enabled
T1LOVF (bit 1)	T1L overflow flag
	0: No overflow flag 1: Overflow flag
T1LIE (bit 0)	T1L interrupt request enabled
	0: Interrupt request disabled 1: Interrupt request enabled

**T1HRUN (bit 7): T1H count control**

Controls count-up start/stop of timer 1 high (T1H). When set to "0", the T1H clock is supplied and counting starts. To stop counting, the bit must be reset to "0". This stops the clock and sends the reload data (T1HR) to T1H.

**T1LRUN (bit 6): T1L count control**

Controls count-up start/stop of timer 1 low (T1L). When set to "0", the T1L clock is supplied and counting starts. To stop counting, the bit must be reset to "0". This stops the clock and sends the reload data (T1LR) to T1L.

**T1LONG (bit 5): Timer 1 bit length select**

Specifies the bit length of T1 as 16 or 8 bit.

Setting the bit to "0" selects a 16-bit timer. For using modes 0 and 1, specify "0".

Resetting the bit to "0" selects an 8-bit timer. For using modes 2 and 3, specify "0".

**ELDT1C (bit 4): Pulse generator data update enable control**

Controls whether the comparison data register (T1HC, T1LC) values for generating the pulse signal are sent to the comparator or not.

When set to "0", the values are sent to the comparator and updated to new pulse generator data.

When reset to "0", the data are not updated and the same pulse generator data are output.

To update both 8-bit counters at the same time, reset this flag, set the counter values, and then set the flag again. This will update both 8-bit counters at the same time.

**T1HOVF (bit 3): T1H overflow flag**

This flag is set when T1L overflow has occurred. If there is no overflow, the flag does not change.

This flag must be reset by the T1H interrupt processing routine or another routine of the application.

**T1HIE (bit 2): T1H interrupt request enable control**

Enables or disables interrupt request generation at T1H overflow.

When set to "0", the interrupt generated by T1H overflow is accepted and the interrupt vector 002BH is called. When reset to "0", the interrupt is not accepted and the interrupt processing routine is not called.

**T1LOVF (bit 1): T1L overflow flag**

This flag is set when T1L overflow has occurred. If there is no overflow, the flag does not change.

Regardless of the T1 bit length, the flag is set when overflow occurs at T1L.

This flag must be reset by the T1L interrupt processing routine or another routine of the application.

**T1LIE (bit 0): T1L interrupt request enable control**

Enables or disables interrupt request generation at T1L overflow.

When set to "0", the interrupt generated by T1L overflow is accepted and the interrupt vector 002BH is called.

- Caution:**
- The overflow flags (T1HOVF, T1LOVF) must be reset to "0" by the application.
  - When using the 16-bit mode, the clock can be the cycle clock or the cycle clock divided by 2.  
Ttc = Tcyc: T1HRUN=1, T1LRUN=1, T1LONG=1  
Ttc = 1/2Tcyc: T1HRUN=0, T1LRUN=1, T1LONG=1  
Ttc is the clock cycle
- 

### Timer 1 low register (T1L)

The timer 1 low register is an 8-bit timer. It uses the cycle clock or the cycle clock divided by 2.

When T1L overflow occurs, the T1LR value is transferred and the T1L overflow flag is set.

In modes 1 and 3, this is used for pulse signal generation.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
T1L	11BH	R	T1L7	T1L6	T1L5	T1L4	T1L3	T1L2	T1L1	T1L0
Reset			0	0	0	0	0	0	0	0

### Timer 1 low reload register (T1LR)

This is the reload register for timer 1 low (T1L).

Each time a T1L overflow occurs, and whenever T1LRUN=0 applies, the reload register value is loaded into T1L.

In modes 1 and 3, this is used for pulse signal generation.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
T1LR	11BH	W	T1LR7	T1LR6	T1LR5	T1LR4	T1LR3	T1LR2	T1LR1	T1LR0
Reset			0	0	0	0	0	0	0	0

T1L and T1LR are at the same address. T1L is read-only, and T1LR is write-only.

- Caution:** When a bit operation instruction or the INC, DEC, or DBNZ instruction is used on a write-only register, a bit other than the specified bit will be set.  
For T1LR, use the following instructions:

MOV, MOV @, ST, ST @, POP

---

### Timer 1 low comparator data register (T1LC)

This is the comparator data register for timer 1 low (T1L).

When ELDT1C (bit 4 of T1CNT) is set and T1LONG=0 applies, the next T1L overflow will cause the value of this register to be sent to the pulse generator control circuit (comparator). When T1LONG=1 applies, the next T1H overflow will have the same effect.

When T1LRUN=0, the value of this register is always sent to the pulse generator control circuit.



Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
T1LC	11AH	R/W	T1LC7	T1LC6	T1LC5	T1LC4	T1LC3	T1LC2	T1LC1	T1LC0
Reset			0	0	0	0	0	0	0	0

**Timer 1 high register (T1H)**

This is an 8-bit timer which uses the cycle clock or the T1L overflow (T1LOVF) as clock. At T1H overflow, the T1H overflow flag is set.

In mode 3, this is used for pulse signal generation.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
T1H	11DH	R	T1H7	T1H6	T1H5	T1H4	T1H3	T1H2	T1H1	T1H0
Reset			0	0	0	0	0	0	0	0

**Timer 1 high reload register (T1HR)**

This is the timer 1 high (T1H) reload register.

Each time a T1H overflow occurs, and whenever T1HRUN=0 applies, the reload register value is loaded into T1H.

In mode 3, this is used for pulse signal generation.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
T1HR	11DH	W	T1HR7	T1HR6	T1HR5	T1HR4	T1HR3	T1HR2	T1HR1	T1HR0
Reset			0	0	0	0	0	0	0	0

T1H and T1LH are at the same address. T1H is read-only, and T1LH is write-only.

**Caution:** When a bit operation instruction or the INC, DEC, or DBNZ instruction is used on a write-only register, a bit other than the specified bit will be set.  
For T1LR, use the following instructions:

MOV, MOV @, ST, ST @, POP

**Timer 1 high comparator data register (T1HC)**

This is the comparator data register for timer 1 high (T1H).

When ELDT1C (bit 4 of T1CNT) is set and T1LONG=0 applies, the next T1L overflow will cause the value of this register to be sent to the pulse generator control circuit (comparator). When T1LONG=1 applies, the next T1H overflow will have the same effect.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
T1HC	11CH	R/W	T1HC7	T1HC6	T1HC5	T1HC4	T1HC3	T1HC2	T1HC1	T1HC0
Reset			0	0	0	0	0	0	0	0

## Circuit Configuration and Operation Principles

### Timer 1 mode setting

ModeClock frequencyT1LONGP17FCRP17DDRP1777770Tcyc00yc01102Tcyc\_1/22222222Tcyc10XX30

### Mode 0: 8-bit reload timer x 2 channels

In mode 0, timer 1 functions as an 8-bit reload timer with two channels. The relationship between the timer value and the reload register (T1LR) setting value is as shown below.

Time until T0HOVF is set (1) (decimal) = (256 - T1HR setting value) x Tcyc  
 Time until T0LOVF is set (1) (decimal) = (256 - T1LR setting value) x Tcyc

Tcyc: Cycle clock

When the count control bit (T1HRUN, T1LRUN) is set, counting starts. When it is reset, counting stops, and the contents of the reload register (T1HR, T1LR) are sent to the counter (T1H, T1L).

When the timer 1 (T1H, T1L) overflows, the overflow flag (T1HOVF, T1LOVF) is set, and the contents of the reload register (T1HR, T1LR) are sent to the counter (T1H, T1L).

When both the overflow flag (T1HOVF, T1LOVF) and interrupt request enable flag (T1HIE, T1LIE) are set, the interrupt request is signalled to the interrupt control circuit.

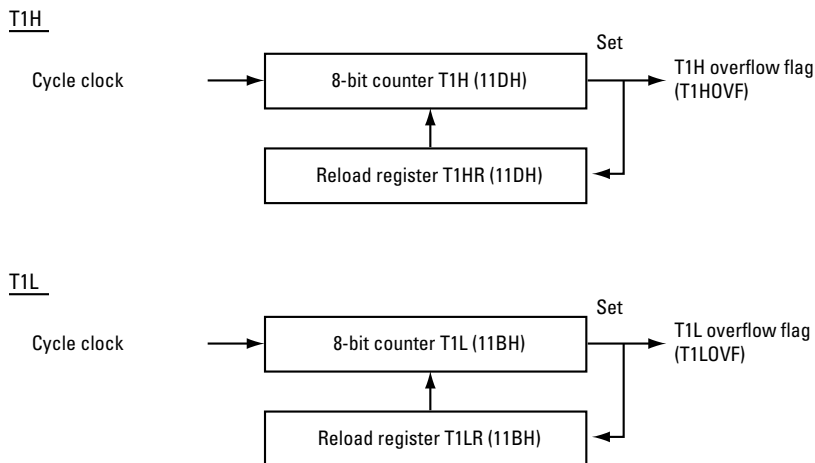
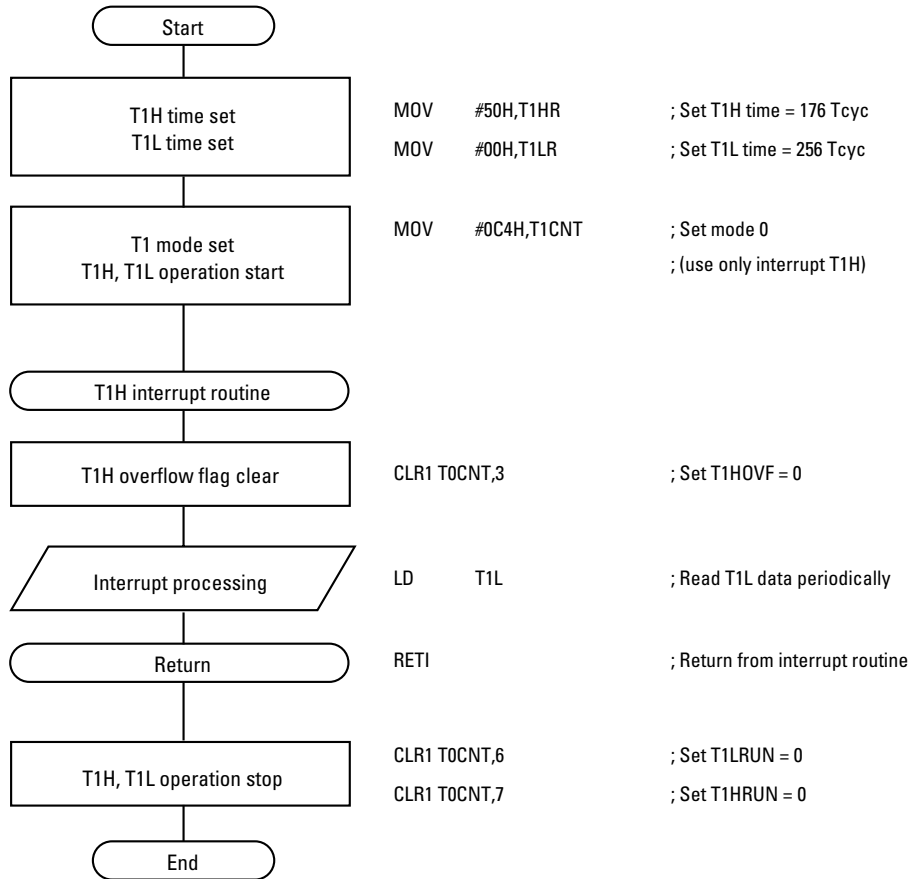


Figure 2.37 8-Bit Reload Timer x 2 Channel Circuit Configuration

**Mode 0 sample program**

• Mode 0 sample program



**Figure 2.38** Flow Chart and Program

**Mode 1: 8-bit reload timer + 8-bit pulse generator**

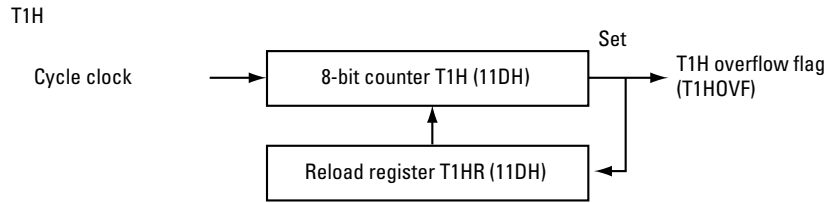
**8-bit reload timer**

The upper 8 bits of timer 0 (T1H) operate as an 8-bit reload timer. The relationship between the timer value and the reload register (T1HR) setting value is as shown below.

$\text{Time until T1HOVF is set (1) (decimal)} = (256 - \text{T1HR setting value}) \times \text{Tcyc}$
--

Tcyc: Cycle clock

Each time T1HOVF is set, the reload register value is sent to the counter T1H. Timer operation continues until the T1H count control bit (T1HRUN) is reset. Operation principles are the same as for mode 0.



**Figure 2.39** Mode 1: 8-Bit Reload Timer (T1H) Block Diagram

## 8-bit pulse generator

The comparator compares the value of the T1L counted up from the reload value by the cycle clock to the value of the comparator data register T1LC. If there is no match ( $T1L \neq T1LC$ ), "0" is output. If there is a match ( $T1L = T1LC$ ), "1" is output. This continues until T1L overflow is generated.

The pulse signal cycle is determined by the reload register T1LR. The relationship between the counter value and the pulse output waveform is shown in Fig. below.

The pulse output waveform is determined by the comparison data register T1LC and the reload register T1LR. When the comparison data register T1LC is rewritten, there will be a pulse cycle delay until the pulse output waveform reflects the new data.

Whenever T1L overflows, the T1L overflow flag (T1LOVF) is set.

The pulse output signal related equations are shown below.

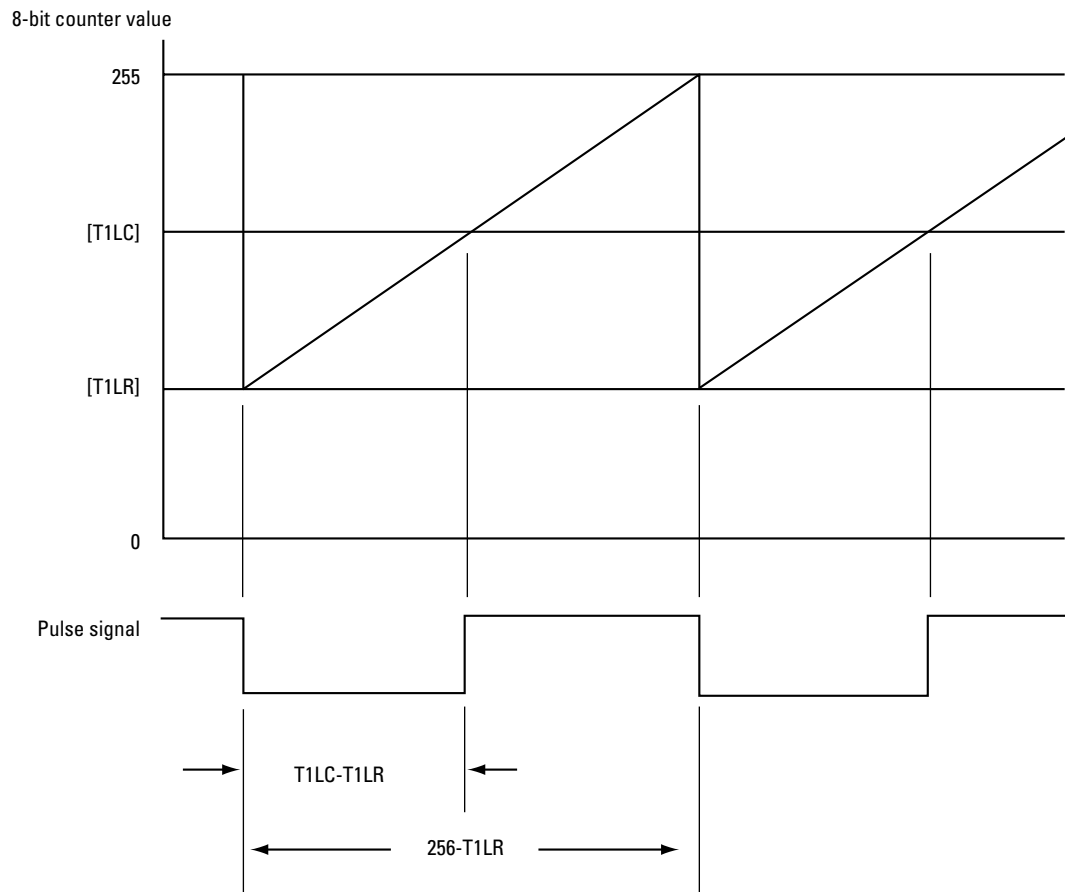
<p>Pulse output signal "L" level pulse width (decimal) = <math>(T1LC \text{ setting value} - T1LR \text{ setting value}) \times T_{cyc}</math> Pulse output signal cycle (decimal) = <math>(256 - T1LR \text{ setting value}) \times T_{cyc}</math></p>
---

T<sub>cyc</sub>: Cycle clock

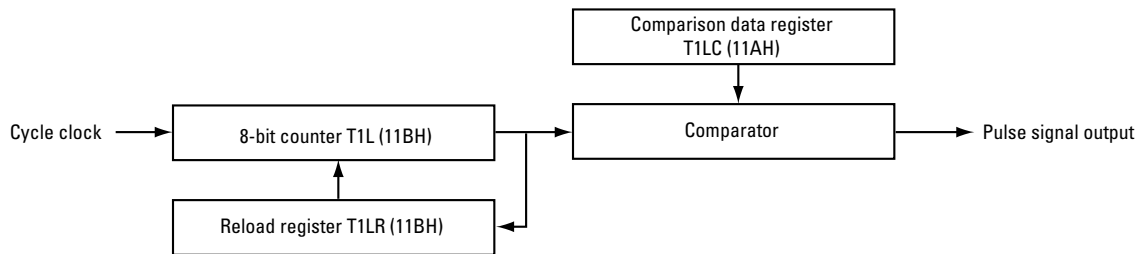
---

**Caution:** Make sure that  $T1LC \geq T1LR$  applies

---



**Figure 2.40** Counter Value and Pulse Generator Output Waveform



**Figure 2.41** Mode 1: 8-Bit Pulse Generator Block Diagram

## Mode 1 sample program

• Mode 1 sample program

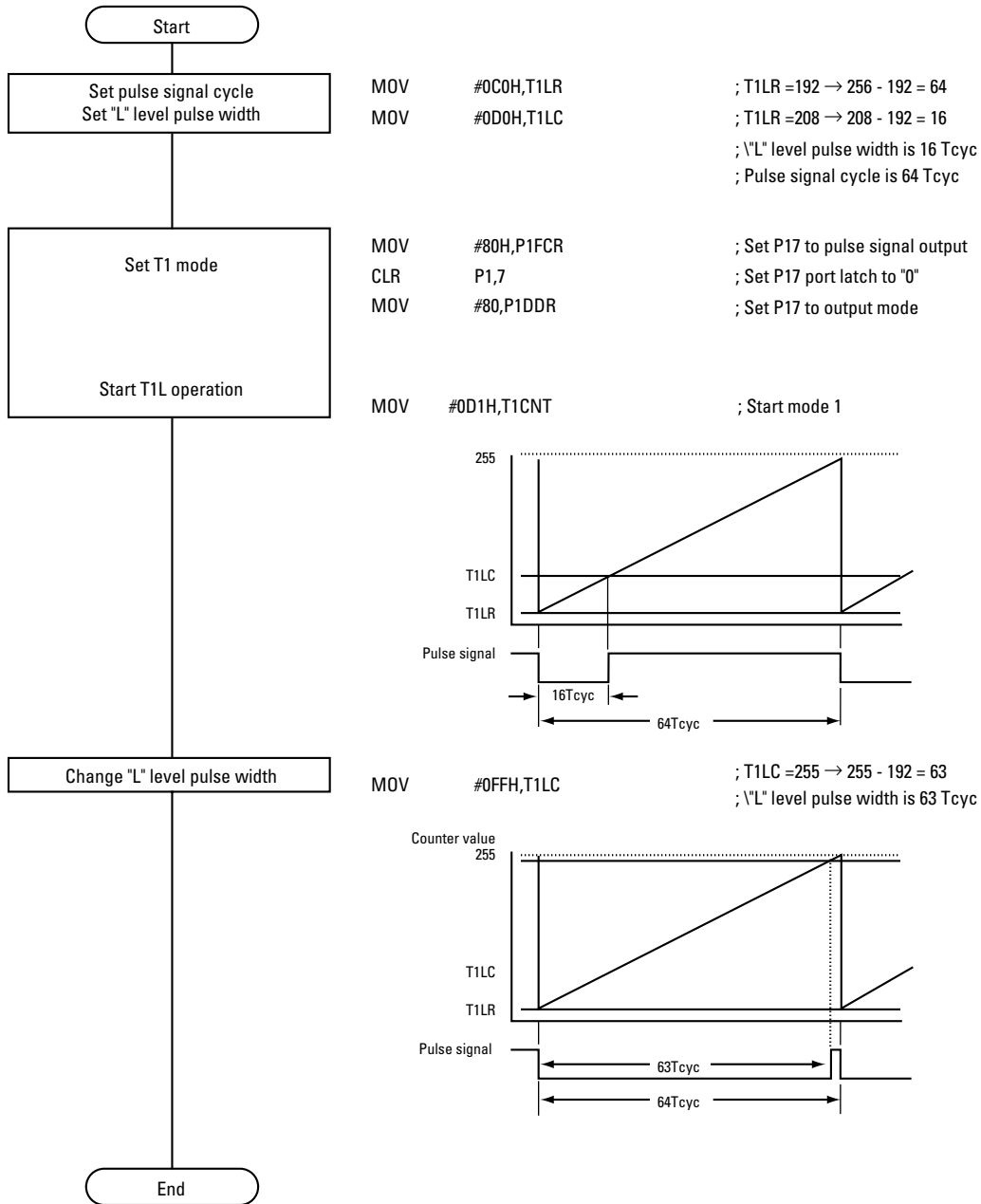


Figure 2.42 Flow Chart and program

**Mode 2: 16-bit reload timer**

To operate the timer as 16-bit reload timer, T1LRUN and T1LONG must be set together. This should be done with the MOV instruction.

The T1L clock (Ttc) can be either the cycle clock (Tcyc) or half the cycle clock (1/2Tcyc). The setting is made as follows.

Ttc = Tcyc:	T1HRUN = 1, T1LRUN = 1, T1LONG = 1
Ttc = 1/2Tcyc:	T1HRUN = 0, T1LRUN = 1, T1LONG = 1

The relationship between the timer value and the reload register (T1HR, T1LR) is as shown below.

**Caution:** Note that this is different from the timer/counter 0 (T0).

Time until T1HOVF is set (1) (decimal) =
(256 - T1HR setting value) x (256 - T1LR setting value) x Ttc
Time until T1LOVF is set (1) (decimal) = (256 - T1LR setting value) x Ttc

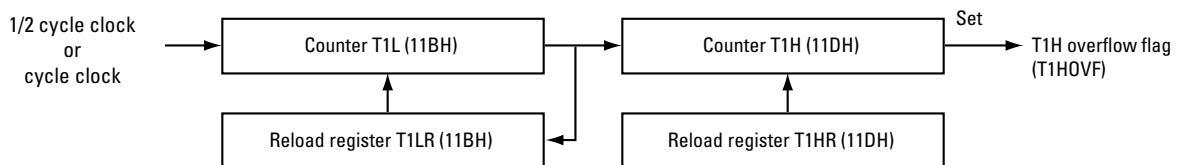
Ttc: T1L clock (Tcyc or 1/2 Tcyc)

At each T1LOVF, the reload data (T1LR) are sent to T1L, and at each T1HOVF, the reload data (T1HR) are sent to T1H. Counting continues until the count control bit is reset. Operation principles are the same as for mode 0.

For reading data from timer 1 (T1), use the following procedure.

```

T1L          LD          T1L ; Read T1L data (1)
↓           ST          020H
T1H          LD          T1H ; Read T1H data
↓           ST          021H
T1L          LD          T1L ; Read T1L (2) data again
↓           BP          T1L,7,DES; When T1L (2) bit 7 is "0"
           BN          020H,7,DES; and T1L (1) bit 7 is "0"
           ST          020H
T1H          LD          T1H ; Read T1H (2)
           ST          021H
           DES:         - next program
    
```



**Figure 2.43** Mode 2: 16-Bit Reload Timer Block Diagram

## Mode 2 sample program

• Mode 2 sample program

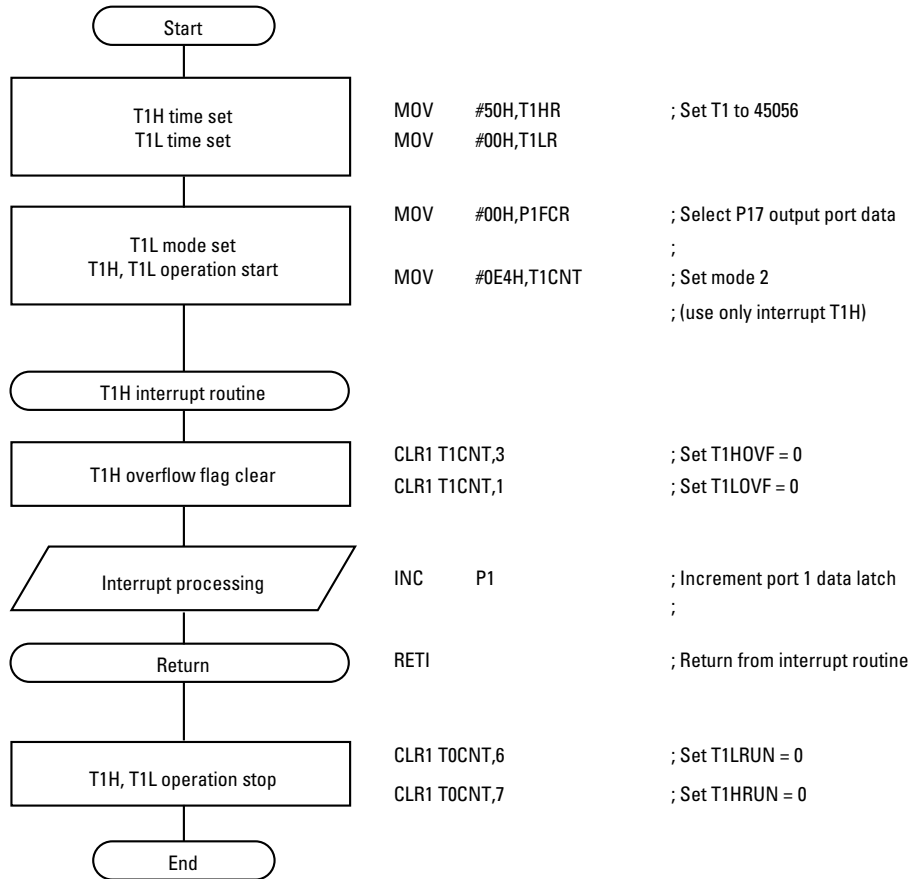


Figure 2.44 Flow Chart and Program

## Mode 3: Variable bit length pulse generator (9 to 16 bits)

In mode 3, timer 1 (T1L, T1H) operates as a variable bit length pulse generator. The range of 9 to 16 bits is determined by T1HR.

To activate the pulse generator, set the bit length of timer 1 to 16 (T1LONG=1) and set the T1L count control bit (T1LRUN). When the 16-bit length is selected, the control bit T1LRUN controls start/stop of all 16 bits. To set the timer 1 control register (T1CNT) bit at the same time, use the MOV instruction.

The pulse generator clock (Ttc) can be either the cycle clock (Tcyc) or half the cycle clock (1/2Tcyc). The setting is made as follows.

Ttc = Tcyc:      T1HRUN = 1, T1LRUN = 1, T1LONG = 1  
 Ttc = 1/2Tcyc:    T1HRUN = 0, T1LRUN = 1, T1LONG = 1

Whenever T1L overflows, the T1L overflow flag (T1LOVF) is set. Similarly, whenever T1H overflows, the T1H overflow flag (T1HOVF) is set. Counting continues until the count control bit is reset.

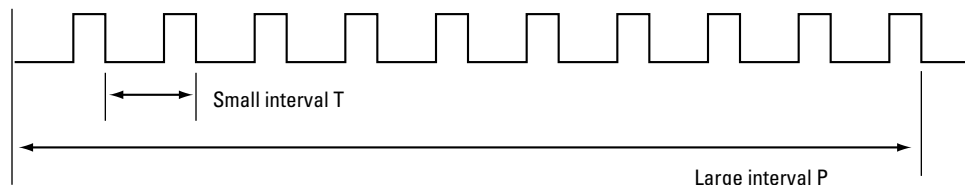


The relationship between the timer value and the reload register (T1HR, T1LR) values is as shown below.

Time until T1HOVF is set (1) (decimal) = $(256 - T1HR \text{ setting value}) \times (256 - T1LR \text{ setting value}) \div Ttc$ Time until T1LOVF is set (1) (decimal) = $(256 - T1LR \text{ setting value}) \div Ttc$
--

Ttc: T1L clock (Tcyc pr 1/2 Tcyc)

An example for a signal output from the pulse output pin P17 in mode 3 is shown in Fig. below.



**Figure 2.45** Mode 3 Pulse Signal Output Waveform

The output signal consists of a repetition of the large interval P which is made up of up to 256 repetitions of the small interval T. The number of repetitions for T can be set with T1HR. The “L” level width in the small interval T can be set with T1LC as in mode 1, and the smallest unit is Ttc. The total “L” level width <math>\leq TL</math> of the large interval P can be set with T1LC and T1HC. The T1HR value limits the data that can be obtained by T1HC.

For details on the relationship between the output waveform and T1HC and T1LC, refer to chapter 17 “Variable Bit Length Pulse Generator” in the appendix.

The relationship between the pulse generator bit length and the value of T1LR and T1HR, as well as the value of T1LC and T1HC is shown below. All T1LR bits are to be set to 00H.

**Table 2.20** Bit Length and T1H/T1L Register

Pulse	Pulse bit length setting (binary)	“L” level pulse width setting (binary)		
Bit length	T1HR value	T1HL value	T1LC value (upper bit)	T1HC value (lower bit)
16	0000 0000	0000 0000	<u>XXXX XXXX</u>	<u>XXXX XXXX</u>
15	1000 0000	0000 0000	<u>XXXX XXXX</u>	<u>XXXX XXX0</u>
14	1100 0000	0000 0000	<u>XXXX XXXX</u>	<u>XXXX XX00</u>
13	1110 0000	0000 0000	<u>XXXX XXXX</u>	<u>XXXX X000</u>
12	1111 0000	0000 0000	<u>XXXX XXXX</u>	<u>XXXX 0000</u>
11	1111 1000	0000 0000	<u>XXXX XXXX</u>	<u>XXX0 0000</u>
10	1111 1100	0000 0000	<u>XXXX XXXX</u>	<u>XX00 0000</u>
9	1111 1110	0000 0000	<u>XXXX XXXX</u>	<u>X000 0000</u>

(X: 0 or 1) X indicates effective bits

For example, if the bit length is 16 bits, the large interval P contains 256 small intervals T, and the following applies:

$$TP = 256 \times T$$

Because the small interval T is  $256 \times T_{tc}$  (cycle clock or 1/2 cycle clock), the following applies:

$$TP = 256 \times 256 \times T_{tc} = 65536 \times T_{tc}$$

The total "L" level additional pulse width STL of the large interval P is set with T1HC.

$$STL+ = [T1HC] \times T_{tc}$$

Because the "L" level width of the small interval T can be set with T1LC, the total "L" level interval width STL is calculated as follows.

$$STL = (256 \times mT1LC) + [T1HC] \times T_{tc}$$

When T1LC = 03H, T1HC = 0B4H

$$STL = (256 \times 03 + 180) \times T_{cyc} = 948 \times T_{tc}$$

The "L" level ratio RL is calculated as follows.

$$RL = STL / TP = 948 / 65536 = \text{approx. } 1.447\%$$

When T1LC = 0FFH, T1HC = 0FFH, the "L" level ratio RL is calculated as follows.

$$RL = STL / TP = 65535 / 65536 = \text{approx. } 99.998\%$$

The relationship between pulse bit length and settable pulse width is as follows.

Large interval P cycle TP

$$TP = 2[\text{bit}] \times T_{tc}$$

Total "L" level pulse width STL in large interval P

$$STL = (2[\text{bit}] = mT1LC) / 256 + [T1HC] \times T_{tc}$$

---

**Note:** T1HC and T1LC are decimal values. [T1HC] is the effective bit value.

---

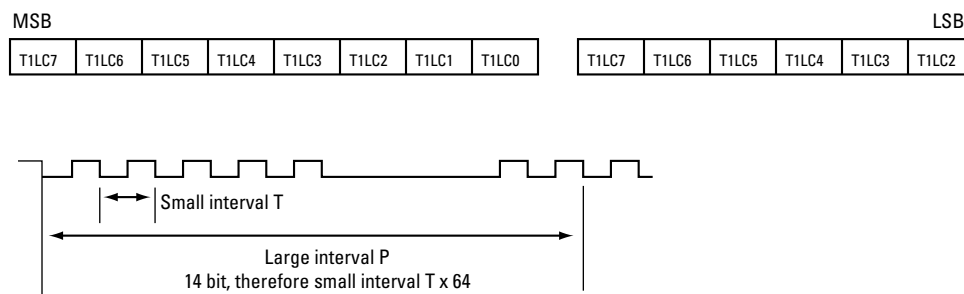
**Table 2.21 Bit Length and Pulse Width and Precision**

Bit length	T1LC		T1HC		_TL		TP[Ttc]	Precision
	min.	max.	min.	max.	min.	max.		
16	0	255	0	255	0	65535	65535	1/65535
15	0	255	0	127	0	32767	32767	1/32767
14	0	255	0	63	0	16383	16383	1/16383
13	0	255	0	31	0	8191	8191	1/8191
12	0	255	0	15	0	4095	4095	1/4095
11	0	255	0	7	0	2047	2047	1/2047
10	0	255	0	3	0	1023	1023	1/1023
9	0	255	0	1	0	511	511	1/511

**Note:** T1HC represents the value for effective bits indicated in the table. For example, with a bit length of 11, bits 7 through 5 are effective. The maximum value therefore is 7.

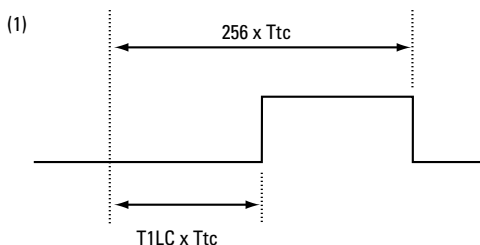
**Example: Setting values (binary) for use as 14-bit pulse generator**

- T1HR value: 1100 0000B
- T1LR value: 0000 0000B
- Pulse generator 14-bit setting value

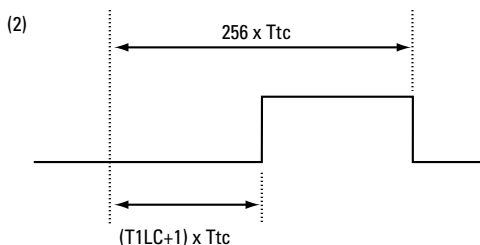


**Figure 2.46** 14-Bit Pulse Generator

In the small interval T, two types of pulses are output. In the large interval P, pulse (1) is (64-T1HC) times, and pulse (2) is output T1HC times.



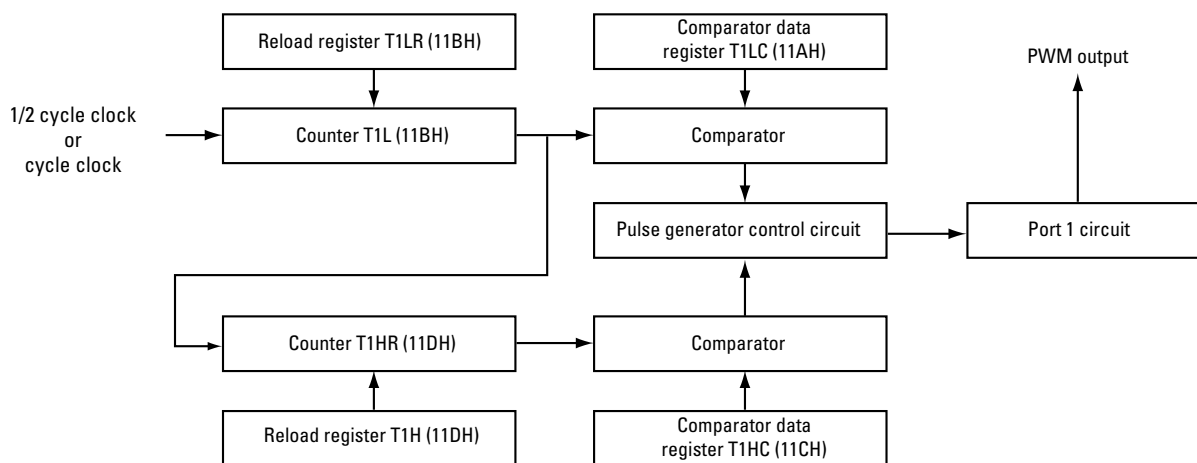
**Figure 2.47**  $T1LC \times Ttc$  pulse



**Figure 2.48**  $(T1LC + 1) \times Ttc$  pulse

For details on the relationship between the output waveform and T1HC and T1LC, refer to chapter 17 “Variable Bit Length Pulse Generator” in the appendix.

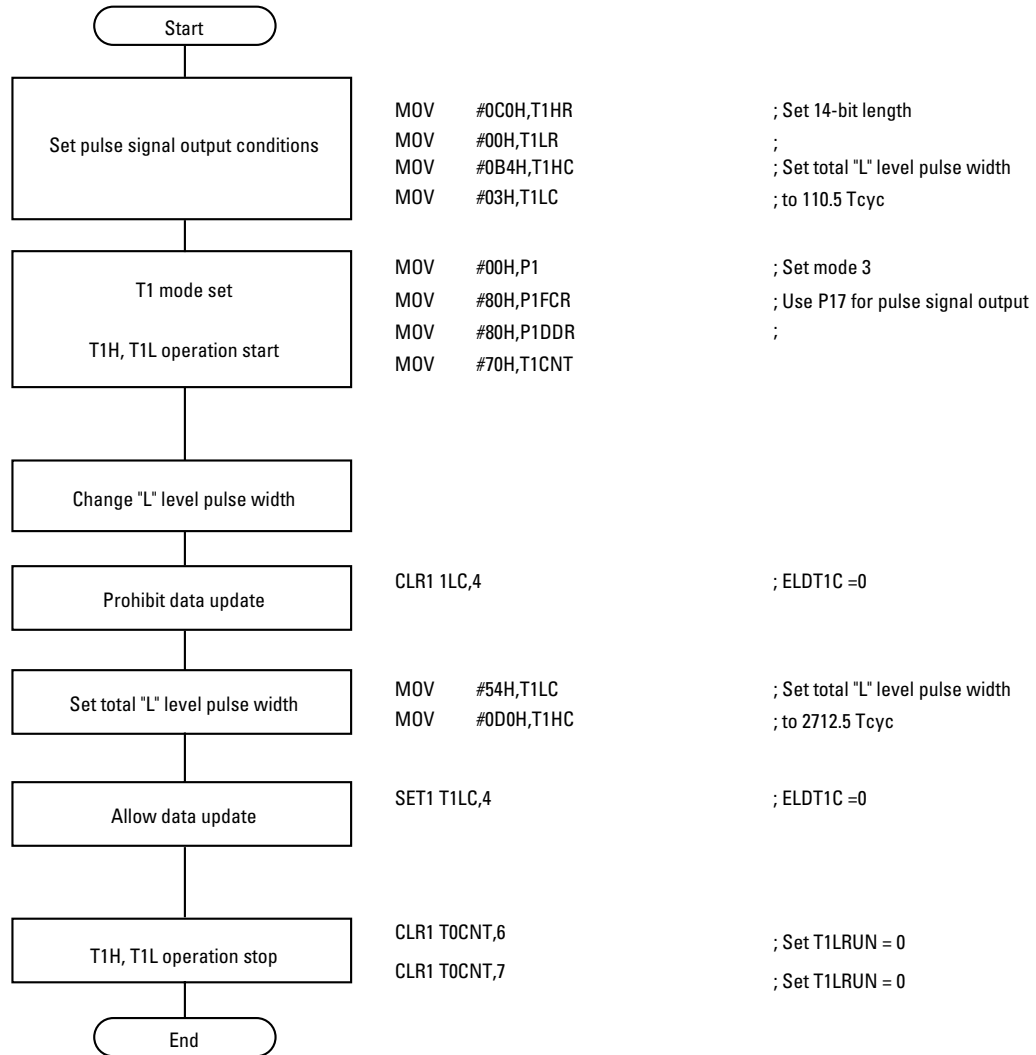
- Caution:**
- To set the “L” level pulse width, use the following procedure.
    - (1) Reset data update enable flag ELDT1C to "0".
    - (2) Rewrite T1LC and T1HC values.
    - (3) Set data update enable flag ELDT1C to "0".
  - The delay between rewriting T1LC and T1HC and the waveform output based on the new data is equal to the interval between setting ELDT1C to "0" and the maximum pulse cycle.
  - For using 16-bit mode, the clock can be either the cycle clock or the cycle clock divided by 2.  
 $T_{tc} = T_{cyc}:T1HRUN=1, T1LRUN=1, T1LONG=1$   
 $T_{tc} = 1/2T_{cyc}:T1HRUN=0, T1LRUN=1, T1LONG=1$
- 



**Figure 2.49** Mode 3: Variable Bit Length Pulse Generator Block Diagram

**Mode 3 sample program**

- Mode 3 sample program



**Figure 2.50** Flow

# Base Timer

The base timer in the VMU custom chip is a 14-bit binary up counter with the following 4 functions.

---

**Caution:** The clock function of the VMU is implemented by counting the interrupts generated in 0.5 second intervals by the base timer. The port 3 interrupt is a level interrupt which is maintained for as long as the user presses a button.

If another timer is used to frequently generate interrupts or to accept the port 3 level interrupt, the internal clock may run slow.

When using the base timer interrupt, call the user-side handler immediately after the label `timer_ex_exit` in `GHEAD.ASM`. The user-side handler must be designed to keep processing time at a minimum, so that the interrupt can be properly processed every 0.5 seconds.

Care must be taken to prevent clock slow-down already when designing an application.

---

- Clock timer
- 14-bit binary up counter
- Fast-forward mode (using 6-bit base timer)

## Functions

### Clock timer

When the 32.768 kHz quartz oscillator is used as count clock for the base timer, a clock with 0.5 second steps can be implemented. The base timer count clock is a quartz oscillator.

### 14-bit binary up counter

By using the 8-bit binary up counter and 6-bit binary up counter in conjunction, a 14-bit binary up counter can be configured. The counters can be cleared by the application.

### Fast-forward mode (using 6-bit base timer)

When the 6-bit timer is used as base timer, and the 32.768 kHz quartz oscillator is used as count clock, a clock with 2 millisecond steps can be implemented. Bit length switching is performed by the base timer control register (BTCR).

### Interrupt generation

When the interrupt request enable bit is set, an interrupt request generated by the base timer will call the register vector 001BH. There are two types of interrupt requests that can be generated by the base timer, referred to as base timer interrupt 0 and base timer interrupt 1.

The following Special Function Registers must be operated to control the base timer.

BTCR, P1, timer 0 functions, interrupt functions

## Circuit Configuration

The base counter configuration is shown in Fig. below.

### 8-bit binary up counter... ①

This is an up counter whose input is selected by the input signal select register (ISL). It generates a 4 kHz/ 2 kHz buzzer output signal. When the counter overflows, it generates a base timer interrupt 1 source. The overflow becomes the clock for the 6-bit binary counter.

### 6-bit binary up counter... ②

This is a 6-bit up counter which uses either the signal selected by the input signal select register (ISL) or 8-bit counter overflow as input. When the counter overflows, it generates a base timer interrupt 0 and 1 source. Input clock switching is performed by the base timer control register BCTR.

### Base timer input clock source... ➔

The quartz oscillator should be selected by the input signal select register (ISL) as base timer clock. Do not use other oscillators.

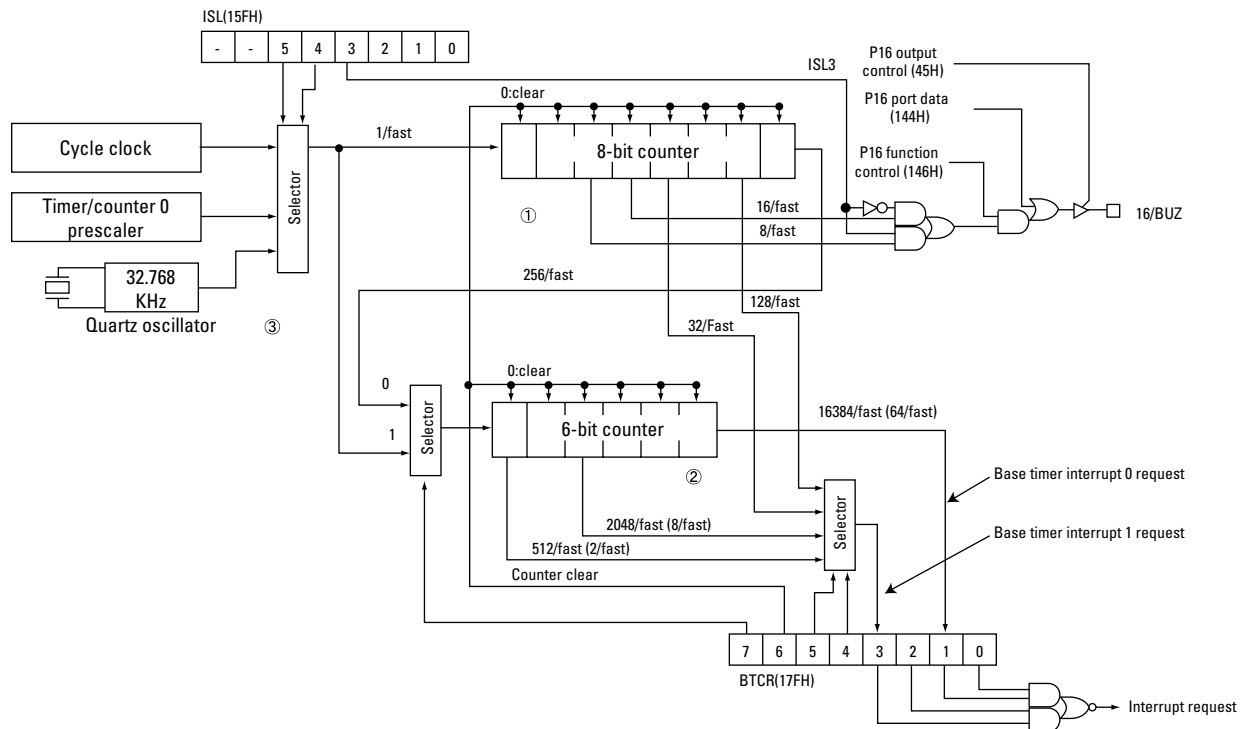


Figure 2.51 Base timer block diagram

## Related Registers

**Table 2.22 Base timer control register (BTCR)**

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
BTCR	17FH	R/W	BTCR7	BTCR6	BTCR5	BTCR4	BTCR3	BTCR2	BTCR1	BTCR0
Reset			0	0	0	0	0	0	0	0

**Caution:** BTCR7, BTCR6, BTCR0 may not be manipulated by an application.  
To operate other bits, be sure to use bit level instructions.

Bit name	Function			
BTCR7 (bit 7)	Base timer interrupt 0 cycle control			
	0: 16384/fBST			
BTCR6 (bit 6)	Base timer operation control			
	1: base timer start			
BTCR5 (bit 5) BTCR4 (bit 4)	Base timer interrupt 1 cycle control			
	BTCR7	BTCR5	BTCR4	
	X	0	0	32/fBST
	X	0	1	128/fBST
	0	1	0	512/fBST
0	1	1	2048/fBST	
BTCR3 (bit 3)	Base timer interrupt 1 source			
	0: Interrupt source disabled 1: Interrupt source enabled			
BTCR2 (bit 2)	Base timer interrupt 1 request enabled			
	0: Interrupt request disabled 1: Interrupt request enabled			
BTCR1 (bit 1)	Base timer interrupt 0 source			
	0: Interrupt source disabled 1: Interrupt source enabled			
BTCR0 (bit 0)	Base timer interrupt 0 request enabled			
	1: Interrupt request enabled			



### **BTCR7 (bit 7): base timer interrupt 0 cycle control 0: fixed**

Specifies the cycle for base timer interrupt 0 source generation.

When set to "0", the cycle is  $16384/f_{BST}$ . In this case, the interval at which the interrupt 0 source is generated for 14-bit counter overflow is  $16384/f_{BST}$ .

When reset to "0", the cycle is  $64/f_{BST}$ . To use the fast- forward mode, set this flag.

---

**Caution:** Because the base timer is used for the clock of the VMU, these registers may never be manipulated by an application.

---

### **BTCR6 (bit 6): base timer operation control 1: fixed**

Starts or stops the base timer count operation.

When set to "0", the count operation starts.

When reset to "0", the count operation stops, and the 14- bit counter is cleared.

---

**Caution:** Because the base timer is used for the clock of the VMU, these registers may never be manipulated by an application.

---

### **BTCR5 - BTCR4 (bits 5 - 4): base timer interrupt 1 cycle control**

Select the cycle for base timer interrupt 1 source generation.

BTCR7	BTCR5	BTCR4	Base timer interrupt 1 cycle
x	0	0	$32/f_{BST}$
x	0	1	$128/f_{BST}$
0	1	0	$512/f_{BST}$
0	1	1	$2048/f_{BST}$

fBST: Input clock frequency

### **BTCR3 (bit 3): base timer interrupt 1 source flag**

This flag is set whenever the base timer interrupt 1 source is generated at the cycle set with BTCR7, BTCR5, and BTCR4. When no interrupt is generated, the flag does not change.

---

**Caution:** This flag must be reset by a suitable interrupt processing routine.

---

### **BTCR2 (bit 2): base timer interrupt 1 request enable control**

Enables or disables the base timer interrupt 1 request.

When set to "0", the base timer interrupt 1 source will generate an interrupt request to interrupt vector 001BH.

When reset to "0", no interrupt request is generated.

### **BTCR1 (bit 1): base timer interrupt 0 source flag**

This flag is set whenever the base timer interrupt 0 source is generated at the cycle set with BTCR7. When no interrupt is generated, the flag does not change.

---

**Caution:** This flag must be reset by a suitable interrupt processing routine.

---

### **BTCR0 (bit 0): base timer interrupt 0 request enable control 0: fixed**

Enables or disables the base timer interrupt 0 request.

When set to "0", the base timer interrupt 0 source will call the interrupt vector 001BH.

When reset to "0", no interrupt request is generated.

---

**Caution:** Because the base timer is used for the clock of the VMU, these registers may never be manipulated by an application.

---

---

**Caution:**

- In fast-forward mode (BTCR7, BTCR5 = 1), do not set both the system clock and the base timer to the quartz oscillator.
- BTCR may occasionally become "0" when BTCR5 and BTCR4 are changed. This is a rare occurrence, but to guard against it, you should save the value of BTCR3 before changing BTCR5 and BTCR4 and then set the value again in BTCR3 after the change.

---

### **Input Signal Select Register (ISL)**

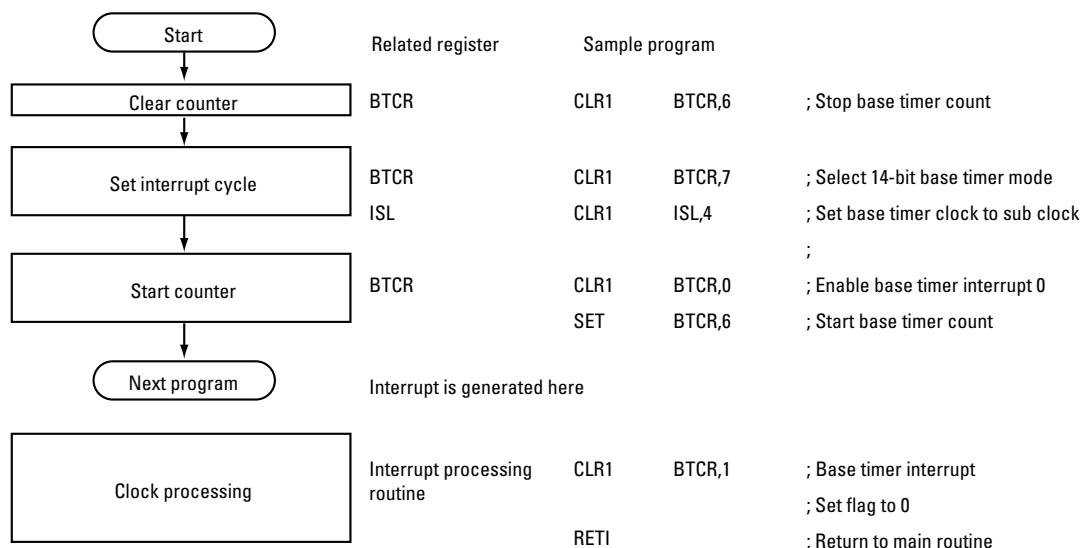
For details, refer to the section "Input Signal Select Register (ISL)" in "Timer/Counter 0 (T0)".

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ISL	15FH	R/W	-	-	ISL5	ISL4	ISL3	ISL2	ISL1	ISL0
Reset			0	0	0	0	0	0	0	0

Bit name	Function		
ISL5 (bit 5) ISL4 (bit 4)	Base timer clock select		
	ISL5	ISL4	
	X	0	Fixed to quartz oscillator
ISL3 (bit 3)	Use prohibited		
	0: fBST/16 (fixed)		
ISL2 (bit 2) ISL1 (bit 1)	Noise filter time constant select		
	ISL2	ISL1	Time constant
	1	1	16Tcyc
	0	1	64 Tcyc
	X	0	1 Tcyc
ISL0 (bit 0)	T0 clock input pin select		
	0: P72/INT2/TOIN pin 1: P73/INT3/TOIN pin		

## Using the Base Timer

• Clock timer



**Figure 2.52** Clock timer

# Serial Interface

The custom chip of the VMU incorporates a 2-channel synchronous serial interface with a data word length of 8 bits. It uses port 1 and allows two VMU units to communicate directly. When connected to the Dreamcast, the interface is automatically switched to the dedicated Dreamcast interface.

---

**Reference:** For information on how to ensure problem-free serial communication, refer to chapter “Serial Communication Precautions” in the appendix.

---

---

**Caution:** Because the dedicated Dreamcast interface (Maple bus mode) also uses port 1, it cannot be used at the same time as the synchronous serial interface. When using the synchronous serial interface, an application must prohibit activation of the dedicated Dreamcast interface.

---

The serial interface has the following main functions and features.

- 2-channel synchronous serial interface
- Selectable transfer clock
- Serial interface SIO0 transfer clock with switchable polarity
- LSB/MSB switchable start sequence
- Switchable operation modes
- Overrun detection
- Transfer bit length control

## Functions and Features

2-channel synchronous serial interface

Two serial interface channels are provided: SIO0 using P10 to P12 and SIO1 using P13 to P15.

Normally, the VMU uses SIO0 as master and SIO1 as slave.

Selectable transfer clock

The following three clock types can be selected. For SIO0, the transfer clock polarity can also be selected.

- Internal clock
- External clock
- Software clock

### Serial interface SIO0 transfer clock with switchable polarity

The polarity of the transfer clock for the serial interface SIO0 can be selected as follows.

- 1) Operation stop, SCK0 = "0"; data output hold
- 2) Operation stop, SCK0 = "0"; data output bit 0 of SBUF0

### LSB/MSB switchable start sequence

Data transfer via the serial interface can start either with the LSB or the MSB. This setting can be made individually for each channel.

### Overrun detection

An error is generated when a clock exceeding 8 bits is received.

### Transfer bit length control

A setting is available to control whether operation stops or continues after 8 bits have been transferred.

### Interrupt

The following Special Function Registers must be operated to control the serial interface.

When the interrupt request enable bit is set, overflow of the octal counter generates an SIO0 or SIO1 interrupt request.

SCON0, SCON1, SBR, SBUF0, SBUF1, P1, P1DDR, P1FCR

---

**Caution:** When data transfer via the serial interface has been carried out, observe the following points.

- (1) Do not make any settings for serial transfer while no other VMU unit is connected. When transfer is completed, be sure to make the settings listed in (3).  
To make serial transfer settings, check the status of port 7 to verify whether another VMU unit is connected.
- (2) When connection of another VMU unit has been verified, make serial transfer settings.  
Use port 7 to verify whether another VMU unit is connected.
- (3) At the end of serial transfer, and when no other VMU unit is connected, establish the following settings.  
SCON0 = 00H  
SCON1 = 00H  
P1FCR = 0BFH  
P1DDR = 0A4H

If serial transfer settings are made while no other VMU unit is connected, normal operation is not assured.

---

## Circuit Configuration

The serial interface configuration is shown in Figures below.

### Shift register... ②

This component consists of two 8-bit shift registers (SBUF0, SBUF1) to operate the specified clock.

### Octal counter...→

Counts the shift clock and detects transfer end.

### Baud rate generator...↪

Consists of an 8-bit register (SBR) for data settings and an 8-bit reload counter. When “internal clock” is selected as transfer clock, the clock data created here are transferred. The baud rate generator is used both by SIO0 and SIO1.

**Reference:** For more information on the internal clock, refer to section on “Serial Transfer Clock”.

### Polarity switcher

Controls the transfer clock polarity before and after serial transfer.

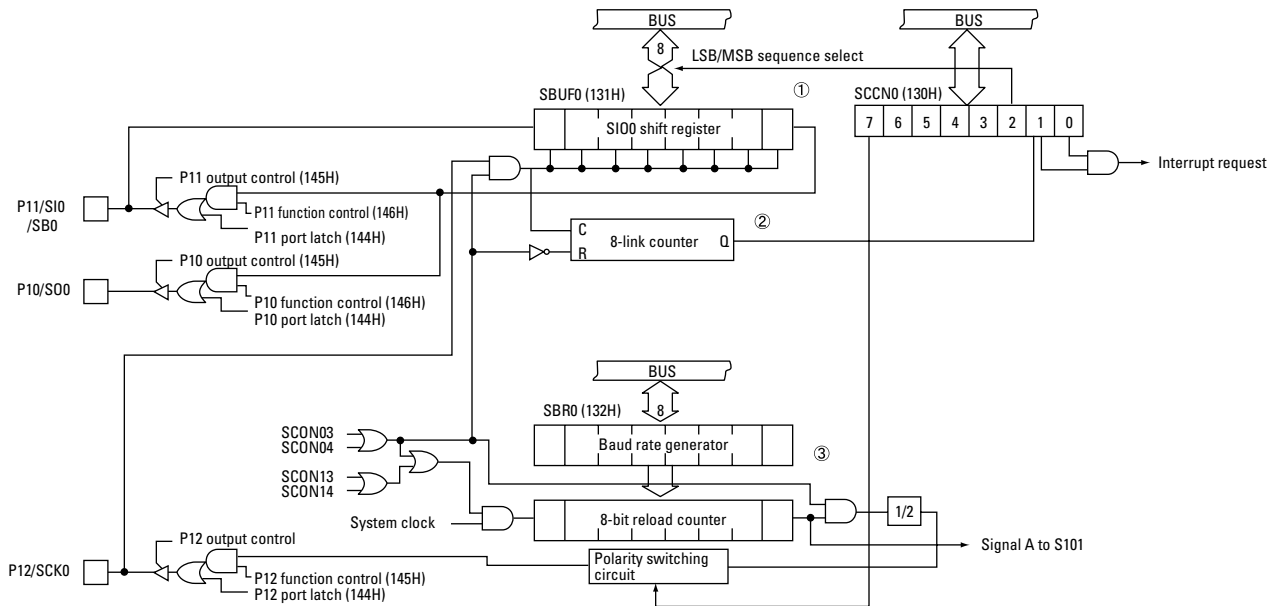


Figure 2.53 Serial interface (SIO0) Block Diagram

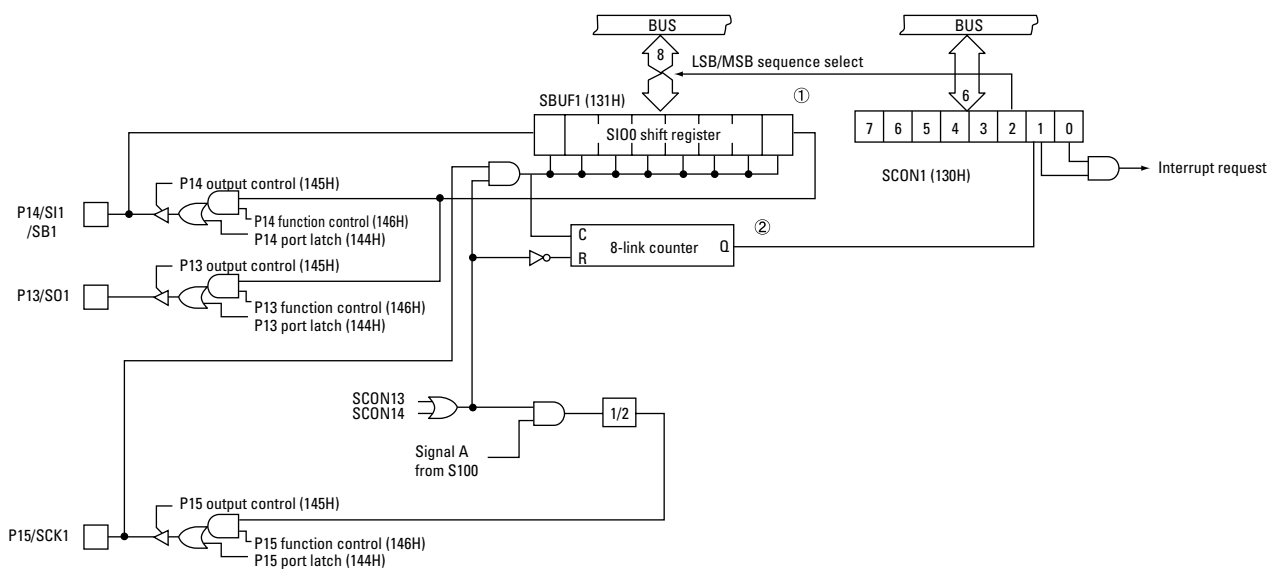


Figure 2.54 Serial interface (SIO1) Block Diagram

## Related Registers

SIO0 control register (SCON0)

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SCON0	130H	R/W	SCON07	SCON06	-	SCON04	SCON03	SCON02	SCON01	SCON00
Reset			0	0	H	0	0	0	0	0

Bit name	Function
SCON07 (bit 7)	Polarity control
	0: at operation stop, SCK0 = 1, maintain data output 1: at operation stop, SCK0 = 0, output data is bit 0 of SBUF0
SCON06 (bit 6)	Overflow flag
	0: No overflow 1: Overflow
SCON04 (bit 4)	Transfer bit length control
	0: 8-bit transfer 1: Continuous transfer
SCON03 (bit 3)	Transfer control
	0: Stop transfer 1: Start transfer
SCON02 (bit 2)	LSB/MSB sequence select
	0: LSB first 1: MSB first
SCON01 (bit 1)	Serial transfer end flag
	0: Transfer in progress 1: Transfer end
SCON00 (bit 0)	Interrupt request enabled
	0: Interrupt request disabled 1: Interrupt request enabled

### SCON07 (bit 7): SCK0 polarity control

Controls the polarity of transfer clock SCK0 used by SIO0.

When set to "0", SCK0 is "0" when SIO0 operation stops, and bit 0 of SBUF0 is output.

When reset to "0", SCK0 is "0" when SIO0 operation stops, and the last transferred data is held at the output.

### SCON06 (bit 6): overflow flag

Detects serial transfer errors in SIO0.

When an 8-bit data transfer is completed (SCON01 has become "0") and the transfer clock is received (falling edge was detected), the flag is set.

During continuous transfer, the overflow flag is set every 8 bits.

---

**Caution:** This flag is not reset automatically. It must be reset by the application.

---



### **SCON04 (bit 4): transfer bit length control**

Switches the SIO0 transfer data bit length to 8 bit continuous (1) or 8 bit (0).

When set to "0", data of 2 byte or more can be sent continuously in 8-bit units.

When reset to "0", only 8 bits of data (1 byte) can be sent.

This flag is not reset after transfer. It must be reset by the application.

### **SCON03 (bit 3): SIO0 operation control**

Starts or stops SIO0 transfer.

When set to "0", 8-bit serial transfer at SIO0 starts. When 8 bits have been transferred, the flag is reset.

When reset to "0", serial transfer at SIO0 stops.

### **SCON02 (bit 2): LSB/MSB start select**

Selects whether data are transferred starting with the MSB or LSB.

When set to "0", the transfer starts with the MSB.

When reset to "0", the transfer starts with the LSB.

---

**Caution:** This flag applies both to sending and receiving data. The setting must match at both ends.

---

### **SCON01 (bit 1): SIO0 transfer end flag**

Detects the end of serial transfer.

The flag is set when a serial transfer of 8 bits is completed.

When this flag is set, and a falling edge of the transfer clock is detected, the overrun flag is set.

---

**Caution:** This flag is not reset automatically. It must be reset by the application.

---

### **SCON00 (bit 0): SIO0 interrupt request enable control**

Enables or disables interrupt request generation at SIO0 transfer end.

When set to "0", the interrupt vector 0033H is called when SIO0 transfer ends.

When reset to "0", no interrupt request is generated.

---

**Caution:** The transfer end flag becomes "0" at the end of 8 bits (1 byte) transfer, regardless of the transfer bit length control setting.  
The overrun flag is only set when overrun was detected. It does not generate an interrupt.

---

## SIO1 control register (SCON1)

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SCON1	134H	R/W	-	SCON16	-	SCON14	SCON13	SCON12	SCON11	SCON10
Reset			H	0	H	0	0	0	0	0

Bit name	Function
SCON16 (bit 6)	Overrun flag
	0: No overrun 1: Overrun
SCON14 (bit 4)	Transfer bit length control
	0: 8-bit transfer 1: Continuous transfer
SCON13 (bit 3)	Transfer control
	0: Stop transfer 1: Start transfer
SCON12 (bit 2)	LSB/MSB sequence select
	0: LSB first 1: MSB first
SCON11 (bit 1)	Serial transfer end flag
	0: Transfer in progress 1: Transfer end
SCON10 (bit 0)	Interrupt request enabled
	0: Interrupt request disabled 1: Interrupt request enabled

### SCON16 (bit 6): overrun flag

Detects serial transfer errors in SIO1.

When an 8-bit data transfer is completed (SCON11 became "0") and the transfer clock is received (falling edge was detected), the flag is set.

During continuous transfer, the overflow flag is set every 8 bits.

---

**Caution:** This flag is not reset automatically. It must be reset by the application.

---

### **SCON14 (bit 4): transfer bit length control**

Switches the SIO1 transfer data bit length to 8 bit continuous (1) or 8 bit (0).

When set to "0", data of 2 byte or more can be sent continuously in 8-bit units.

When reset to "0", only 8 bits of data (1 byte) can be sent. When the 8-bit transfer is completed, the transfer end flag (SCON11) is set.

---

**Caution:** This flag is not reset after transfer. It must be reset by the application.

---

### **SCON13 (bit 3): SIO1 operation control**

Starts or stops SIO1 transfer.

When set to "0", 8-bit serial transfer at SIO1 starts. When 8 bits have been transferred, the flag is reset.

When reset to "0", serial transfer at SIO1 stops.

### **SCON12 (bit 2): LSB/MSB start select**

Selects whether data are transferred starting with the MSB or LSB.

When set to "0", the transfer starts with the MSB.

When reset to "0", the transfer starts with the LSB.

---

**Caution:** This flag applies both to sending and receiving data. The setting must match at both ends.

---

### **SCON11 (bit 1): SIO1 transfer end flag**

Detects the end of serial transfer.

The flag is set when a serial transfer of 8 bits is completed.

When this flag is set, and a falling edge of the transfer clock is detected, the overrun flag is set.

---

**Caution:** This flag is not reset automatically. It must be reset by the application.

---

### **SCON10 (bit 0): SIO1 interrupt request enable control**

Enables or disables interrupt request generation at SIO1 transfer end.

When set to "0", the interrupt vector 003BH is called when SIO1 transfer ends.

When reset to "0", no interrupt request is generated.

---

**Caution:** The transfer end flag becomes "0" after 8 bits (1 byte) have been transferred, regardless of the transfer bit length control setting.

---

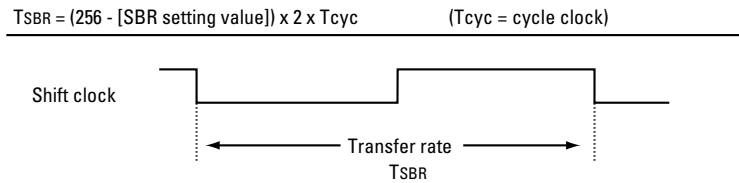
The overrun flag is only set when overrun was detected. It does not generate an interrupt.

---

## Baud rate generator register (SBR)

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SBR	132H	R/W	SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0
Reset			0	0	0	0	0	0	0	0

When the internal clock is used as transfer clock, this register sets the transfer rate. The value is common to both SIO0 and SIO1. The transfer rate  $T_{SBR}$  can be obtained by the following equation.



**Figure 2.55** SIO0, SIO1 transfer rate

## Serial buffer 0 (SBUF0)

Stores transfer data (8 bits) from SIO0.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SBUF0	131H	R/W	SBUF07	SBUF06	SBUF05	SBUF04	SBUF03	SBUF02	SBUF01	SBUF00
Reset			0	0	0	0	0	0	0	0

## Serial buffer 1 (SBUF1)

Stores transfer data (8 bits) from SIO1.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SBUF1	135H	R/W	SBUF17	SBUF16	SBUF15	SBUF14	SBUF13	SBUF12	SBUF11	SBUF10
Reset			0	0	0	0	0	0	0	0

## Dedicated Dreamcast interface circuit

In addition to the serial interface described above, port 1 also operates as an input/output port for the dedicated Dreamcast interface. It is not possible to use the dedicated Dreamcast interface and the serial interface simultaneously.

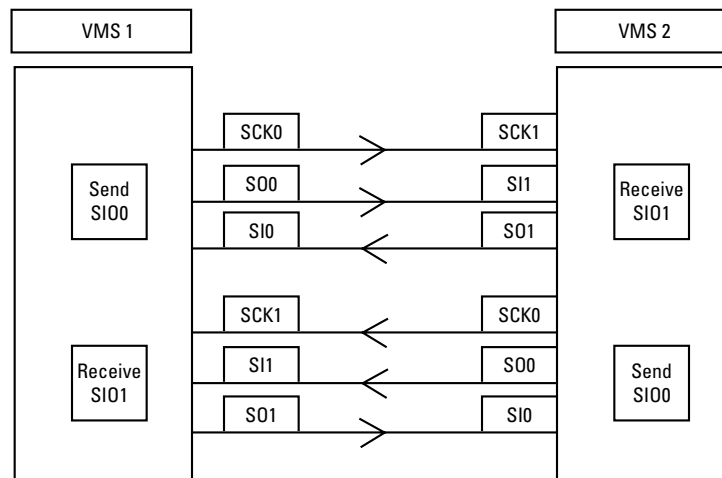
## Serial Interface Operation

Transfer via the serial interface starts when the transfer control bit (SCON03, SCON13) or transfer bit length select bit (SCON04, SCON14) is set. The transfer mode can be the Maple bus mode or normal mode, but applications can only use the normal mode.

### Normal mode

Data transfer uses two data lines and one clock line. The data lines are S1 (dedicated input) and S0 (dedicated output). This mode follows conventional transfer principles and is suited for communication with a specific device.

**Caution:** For communication between two VMU units, use normal mode.



**Figure 2.56** Connection of two VMU units

The transfer mode can be specified by operating the Special Function Register assigned to port 1 (refer to below). The mode can be specified separately for SIO0 and SIO1.

### Serial transfer timing

The shift registers are synchronized to the falling edge of the serial clock SCK0 and SCK1, and data from the shift registers are output at the S00 and S01 pins. At the rising edge of the serial clock, the data input from pins S10 and S11 are read into the shift registers.

## Operation Mode Settings

### Normal mode

The output pins or port latch data for the transfer clock used as internal clock must be reset. The following pins are used in normal mode.

**Table 2.23 Pins Used in Normal Mode**

Mode	SI00	SI01
Input pin	P11/SI0/SB0	P14/SI1/SB1
Output pin	P10/S00_P11/SI0/SB0	P13/S01_P14/SI1/SB1_
Transfer clock	P12/SCK0	P15/SCK1

---

**Caution:** Tcyc before starting transfer, SCKn is set to "0". At less than 1 Tcyc, correct data will not be output.

---

**Table 2.24 Port 1 Settings for SI00 (Special Function Registers)**

Pin	Function	Special function register value
P11/SI0/SB0 P10/S00	Receive Send	P11DDR = 0 P10 = 0 P10DDR = 1 P10FCR = 1
P11/SI0/SB0 P10/S00	Receive General input/output	P11DDR = 0 P10FCR = 0
P12/SCK0	Internal clock	P12 = 0 P12DDR = 1 P12FCR = 1

---

**Note:** The software clock is programmed to alternately write "0" and "0" to the port (P12), and the output is used as transfer clock.

---

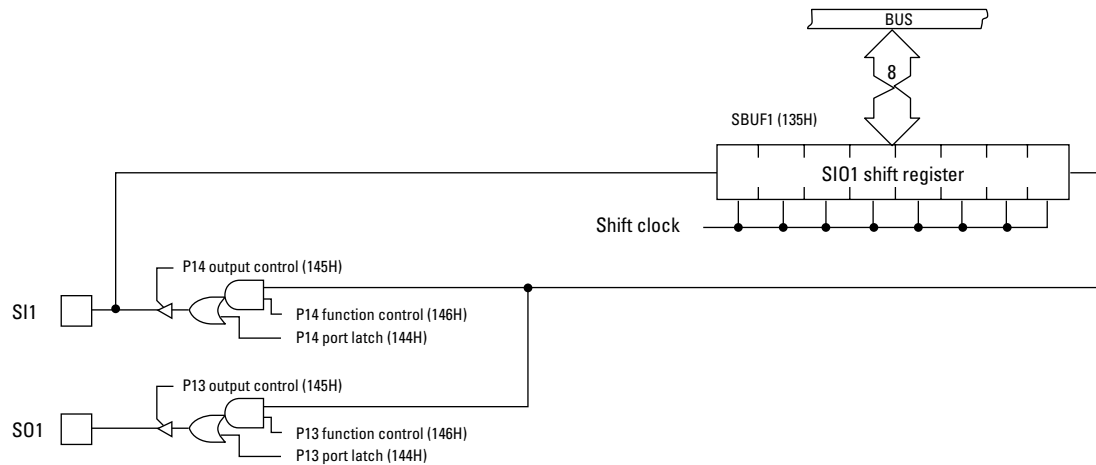
**Table 2.25 Port 1 Settings for SI01 (Special Function Registers)**

Pin	Function	Special function register value
P14/SI11/SB1	Receive	P14DDR = 0
P13/S01	General input/output	P13FCR = 0

---

**Note:** The software clock is programmed to alternately write "0" and "0" to the port (P15), and the output is used as transfer clock.

---



**Figure 2.57** Normal mode signal path (SIO1 example)

**Caution:** To set Pn to “output”, PnFCR must be set to "0" before PnDDR. If PnDDR is set first, "0" may be output from Pn when PnDDR is set. This applies both to SIO0 and SIO1.

## Serial transfer clock

The serial transfer clock (shift clock) uses the P12/SCK0 pin for SIO0 and the P15/SCK1 pin for SIO1. According to the applied circuit specifications, one of the following three clock types can be selected for SIO0 and SIO1 separately. For SIO0, the transfer clock polarity can also be selected.

- Internal clock
- External clock
- Software clock

### Internal clock

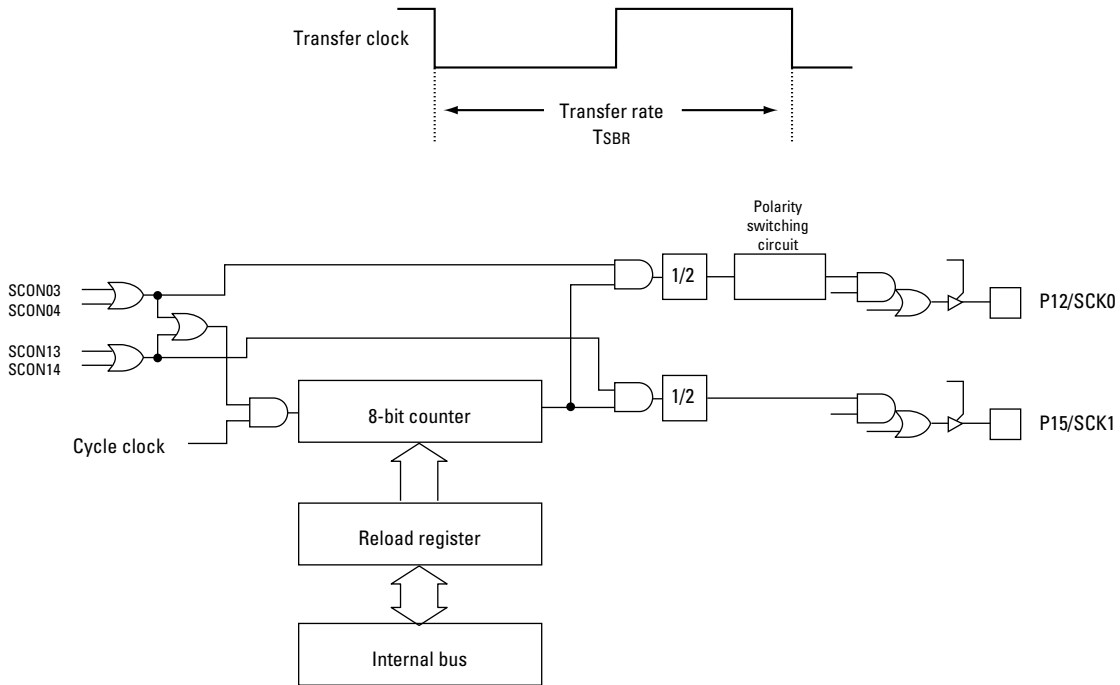
Normally, the internal clock is used for serial transfer. The dedicated serial baud rate generator (SBR) integrated in the VMU custom chip generates the transfer clock which is supplied to the SIO0 and SIO1 circuitry.

When the serial interface is driven with the internal clock, the baud rate generator must be activated. When this is done, the serial transfer clock will be output from the serial interface clock pin (P12/SCK0, P15/SCK1).

The relationship between the transfer rate and the baud rate generator setting is as shown below. The setting values are decimal.

$$TSBR = (256 - [\text{SBR setting value}]) \times 2 \times T_{\text{cyc}} \quad (T_{\text{cyc}} \text{ is the cycle clock})$$

$$TSBR = (256 - [SBR \text{ setting value}]) \times 2 \times T_{cyc} \quad (T_{cyc} = \text{cycle clock})$$



**Figure 2.58** Baud Rate Generator Configuration Diagram

**Caution:** To set Pn to “output”, PnFCR must be set to “0” before PnDDR. If PnDDR is set first, “0” may be output from Pn when PnDDR is set. This applies both to SIO0 and SIO1.

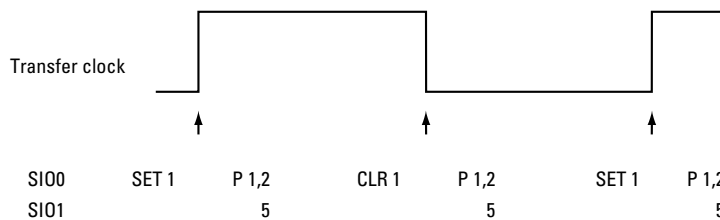
**External clock**

The VMU custom chip can perform serial transfer using an externally supplied clock.

**Software clock**

Using a program to alternately output “0” and “0” from ports P12/SCK0 and P15/SCK1, the output can be used as serial transfer clock.

Clock generation example



**Figure 2.59** Clock Generation Example



To use this type of transfer clock, appropriate settings must be made first for ports P12/SCK0 and P15 SCK1.

**Table 2.26 Transfer Clock Settings**

Pin	Function	Special function register value
P12/SCK0	Internal clock	P12 = 0 P12DDR = 1 P12FCR = 1
	External clock	P12DDR = 0
	Software clock	P12 = 0/1 P12DDR = 1 P12FCR = 0
P15/SCK1	Internal clock	P15 = 0 P15DDR = 1 P15FCR = 1
	External clock	P15DDR = 0
	Software clock	P15 = 0/1 P15DDR = 1 P15FCR = 0

- Caution:**
- Serial data and serial clock pulse width must be at least 1/2 the cycle time. This is especially important when using the quartz oscillator or the external clock. For example, when using the 32.768 kHz quartz oscillator, the cycle clock will be 366 ms, requiring a pulse width of at least 183 ms.
  - When outputting the serial clock from port 1, the port 1 registers must be set in the order shown below, otherwise correct operation is not assured.

- 1) P1FCR setting
- 2) P1DDR setting
- 3) SCONn setting (transfer control bit setting)

## Serial Transfer Timing

During serial transfer, the transfer clock SCK0 of SIO0 (when SCON07 = 0) and SIO1 outputs a "High" level (SCK0 = 1) before and after operation. The last transferred data is held at the output.

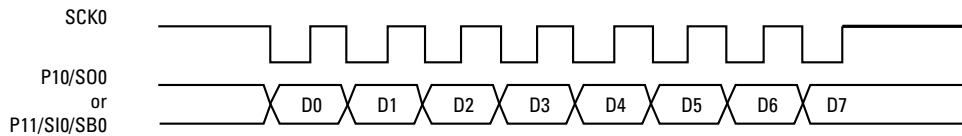
When SCON07 = 1, the transfer clock SCK0 of SIO0 outputs a "low" level (SCK0 = 0) before and after operation. Bit 0 (SBUF00) of the serial buffer 0 (SBUF0) is held at the output (refer to Table 4-40). In SIO1, polarity switching is not possible.

## SIO0

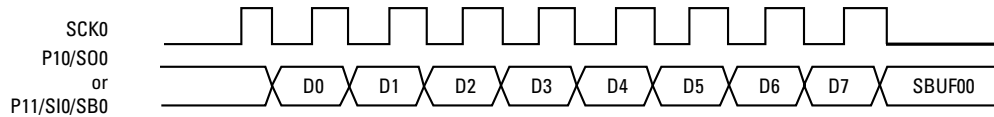
SCON07 = 0	At operation stop, SCK0 = 1, maintain data output
SCON07 = 1	At operation stop, SCK0 = 0, output data is bit 0 of SBUF0

## SIO1

At operation stop, SCK0 = 1, maintain data output
---



**Figure 2.60** Transfer Clock and Output Data (1)



**Figure 2.61** Transfer Clock and Output Data (2)

## LSB/MSB Switchable Start Sequence

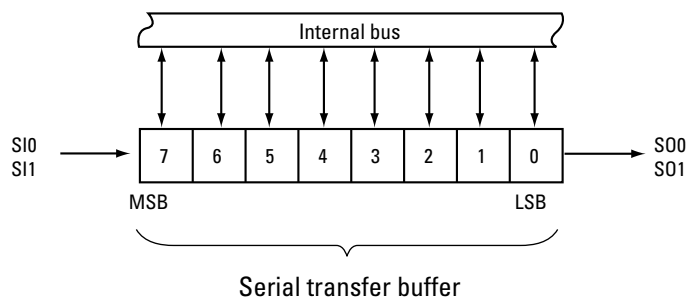
The serial transfer buffer read/write order for data can be set to either LSB → MSB or MSB → LSB.

**Note:** For VMU, either of these is acceptable.

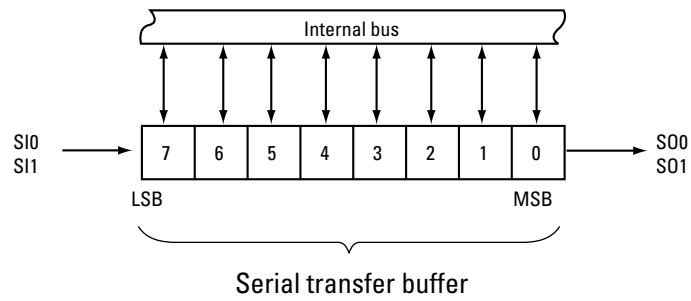
The method used by computers complying to the RS-232C standard is LSB → MSB.

This function allows selection of “LSB-first” or “MSB-first” order for serial data transfer. The setting is made with the serial transfer control register (SCON0, SCON1).

**Caution:** The LSB/MSB order selection must be made before the start of transfer. If the setting is changed after transfer has started, the transfer continues with the original setting.

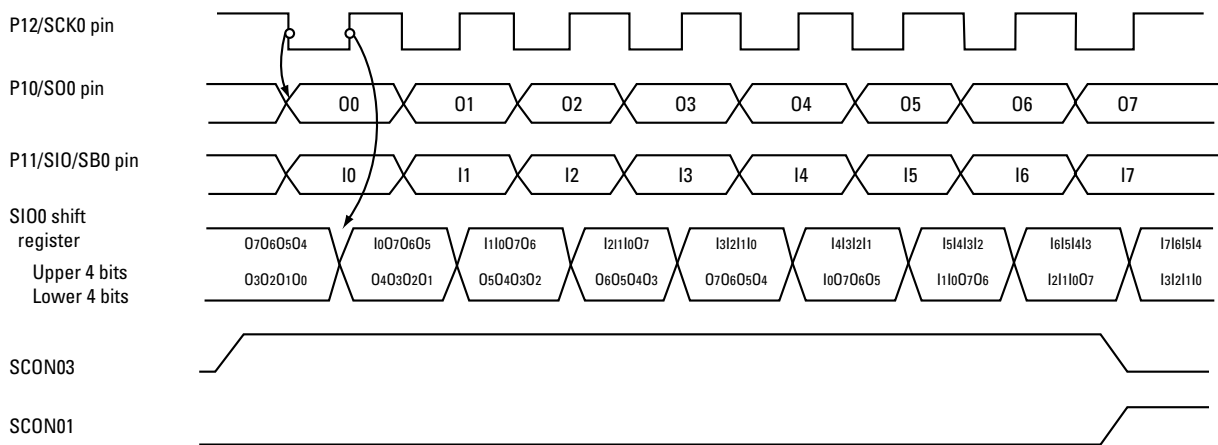


**Figure 2.62** Serial Transfer Buffer and Internal Bus When LSB First Is Selected

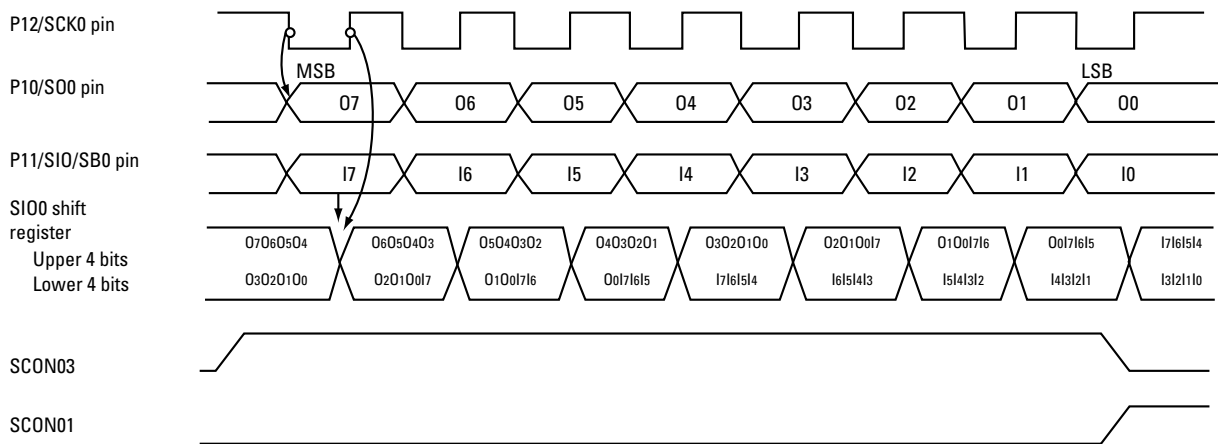


**Figure 2.63** Serial Transfer Buffer and Internal Bus When MSB First Is Selected

The Figures below show the serial transfer send/receive timing in SIO0 when using LSB first and MSB first.



**Figure 2.64** Serial Transfer Buffer and Internal Bus When LSB First Is Selected



**Figure 2.65** Serial Transfer Buffer and Internal Bus When MSB First Is Selected

## Overrun Detection

The overrun detection function serves to catch serial transfer errors.

If the interrupt source flag (SCON01, SCON11) is set, the overrun flag (SCON06, SCON16) will be set at the falling edge of the transfer clock.

The normal send timing and overrun timing are shown in Fig. below. At the rising edge of the transfer clock for the 8th data bit, the interrupt source flag (SCON01, SCON11) is set. When the falling edge of the transfer clock is detected in this condition, the overrun detection flag is set.

The overrun flag is only set when overrun is detected. It does not generate an interrupt or have other results.

- Caution:**
- Before checking the overrun flag, wait at least 1/2 transfer clock cycles after the interrupt source flag was set to "0".
  - Also when a transfer mode exceeding 8 bits (continuous transfer mode) was set, the overrun detection function operates in the same way as for 8-bit transfer.

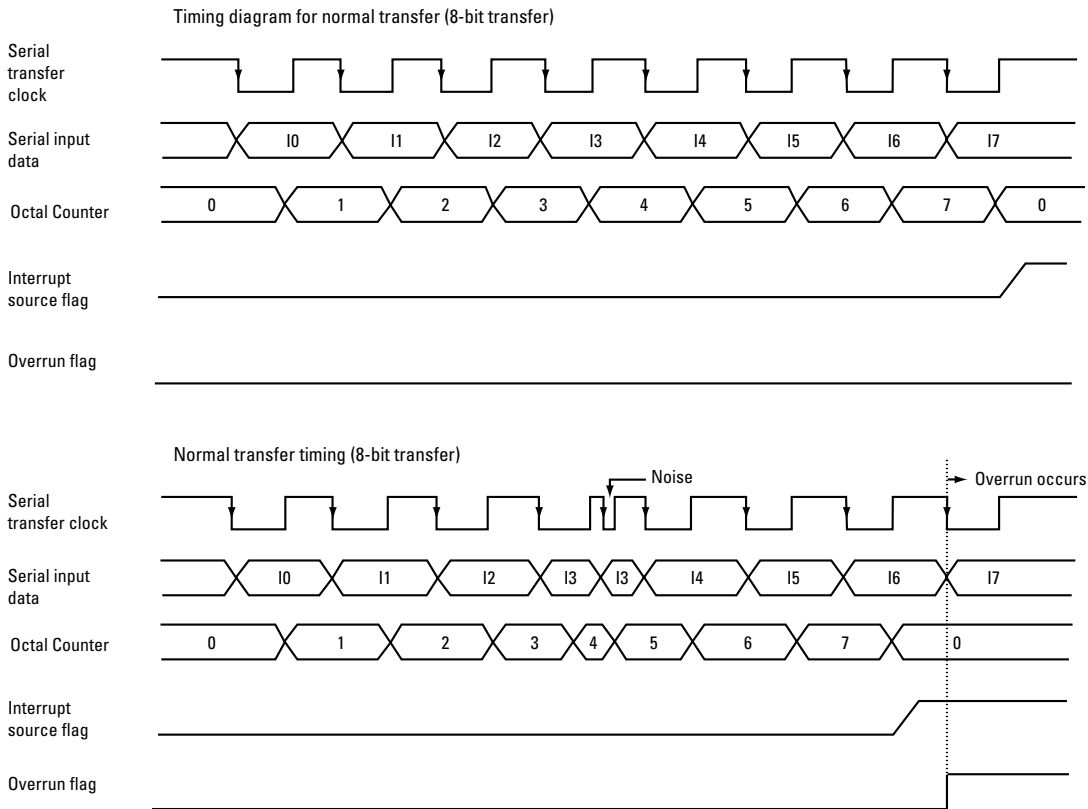


Figure 2.66 Serial Transfer Buffer and Internal Bus When LSB First Is Selected

## Transfer Bit Length Control

When sending more than 8 bits of serial data, continuous transfer can be selected with the transfer bit length control bit SCON04 or SCON14.

- When SCON04 or SCON14 is set, serial transfer starts. This bit is not reset also after 8 bits of data have been transferred.
- The interrupt source flag is set with the same timing as for 8-bit transfer (at 8-bit transfer end).
- The overrun detection bit SCON06 or SCON16 is set at the serial clock falling edge when 8 bits have been exceeded. For information on the timing, see section “Overrun Detection”.
- When the transfer bit length has been set to 8 bits, transfer starts when the transfer control bit SCON03 or SCON13 is set. When 8 bits of data have been transferred, the transfer control bit is reset which in turn causes the interrupt source flag (SCON01, SCON11) to be set. Serial transfer stops automatically.
- When the transfer bit length has been set to continuous, transfer starts when the transfer bit length control bit SCON04 or SCON14 is set. Transfer continues until the bit is reset. The interrupt source flag is set after 8 bits of data have been transferred.

## Sample Program

### SIO0 serial transfer (1) (send) sample program

#### Transfer parameters

- 8-bit transfer
- Transfer data: 038H (8 bits)
- MSB-first
- Falling edge output
- Normal mode
- Internal clock
- Baud rate: 25.6 ms
- System clock: 32.768 kHz quartz oscillator

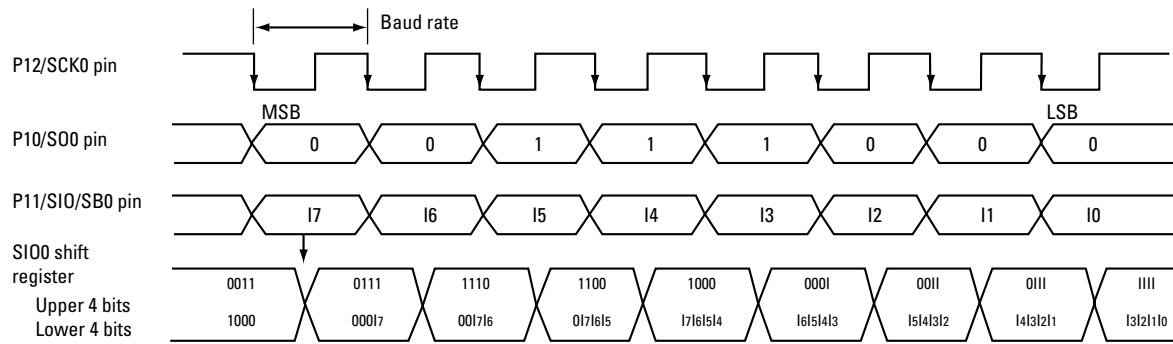
The baud rate equation yields the following

$$TSBR = (256 - [SBR]) \times 2 \times T_{cyc}$$

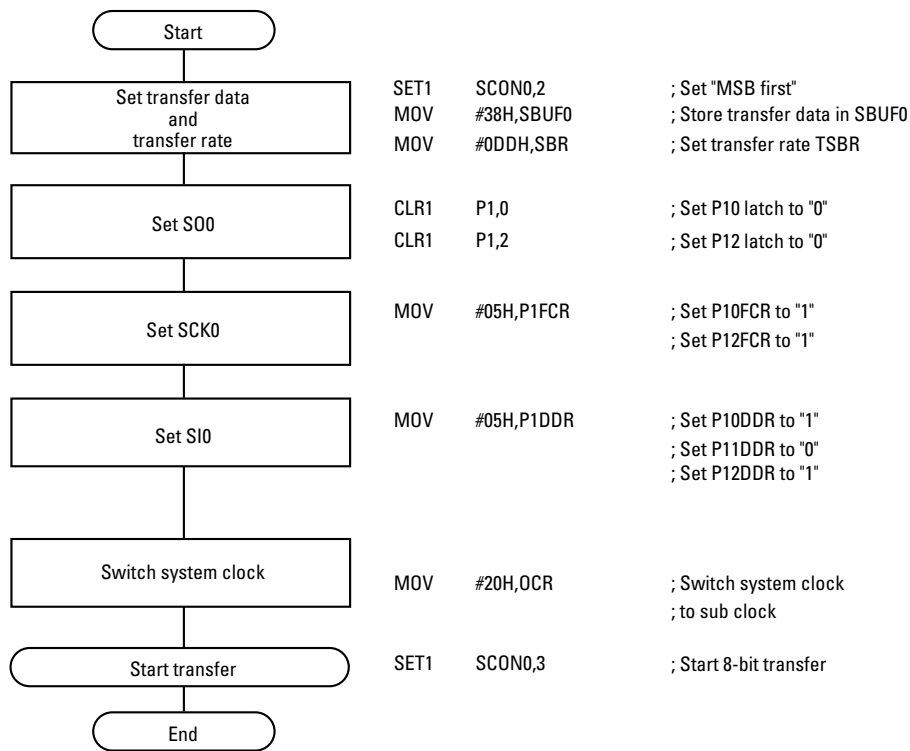
$$[SBR] = 256 - TSBR / (2 \times T_{cyc})$$

Here, the result is  $TSBR = 25.6 \text{ ms}$ ,  $T_{cyc} = 366 \text{ us}$ . The baud rate generator register (SBR) setting value therefore is as follows.

$$\begin{aligned} [SBR] &= 256 - 25600 / (2 \times 366) \\ &= \text{approx. } 221 \text{ (decimal)} \\ &\rightarrow 0DDH \text{ (hex)} \end{aligned}$$



**Figure 2.67** Serial Transfer (1) Timing

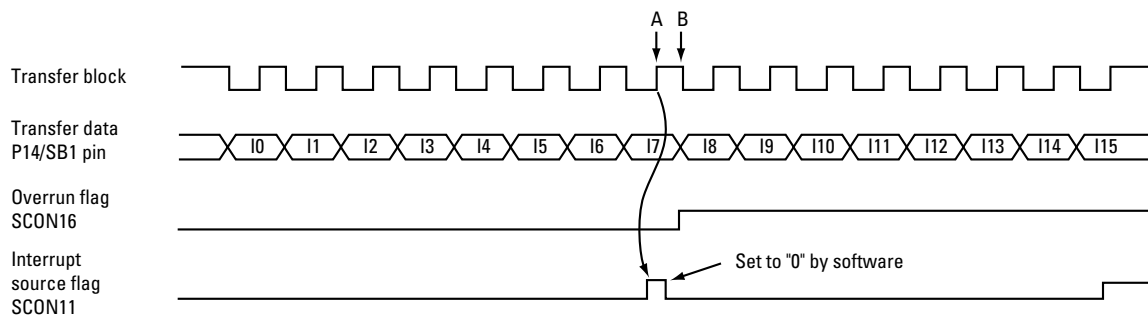


**Figure 2.68** Serial Transfer (Send) Sample Program

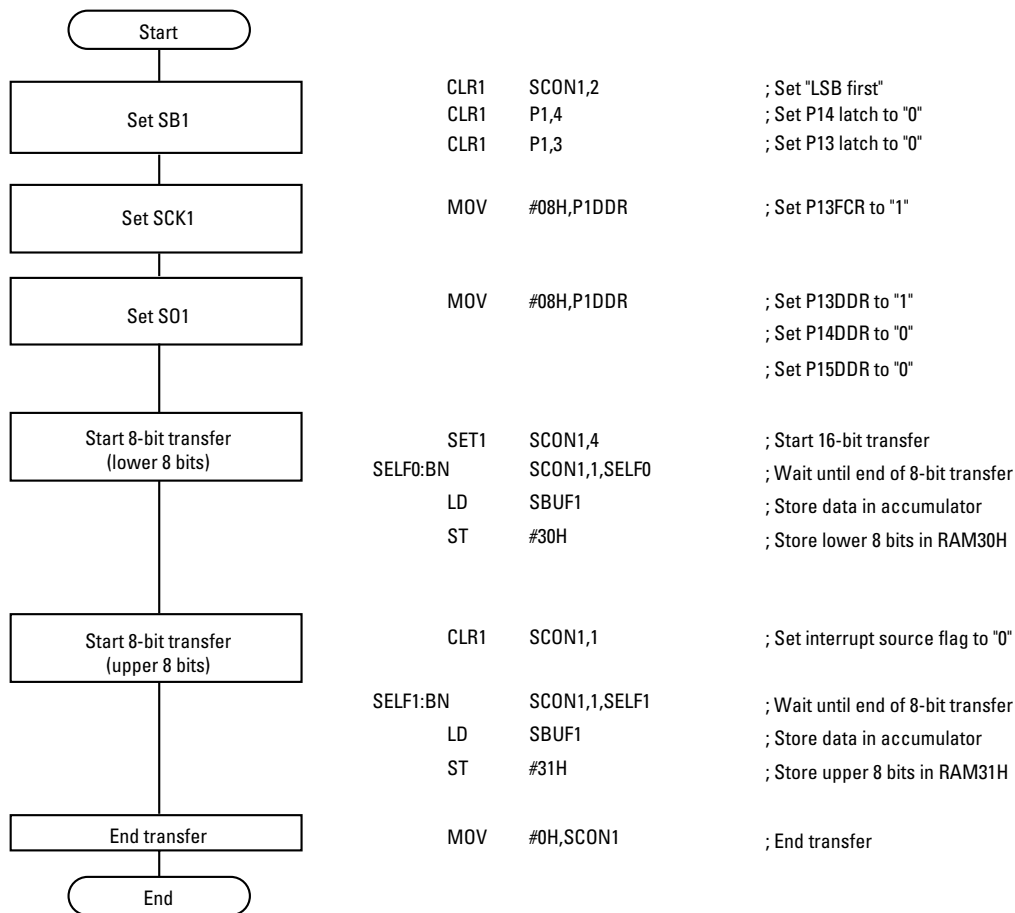
## SIO1 serial transfer (2) (receive) sample program

Transfer parameters

- 16-bit transfer
- LSB-first
- External clock
- Same output data from SO1 as SB1
- Store upper 8 bit of read data at RAM address 031H, and lower 8 bit at RAM address 030H



**Figure 2.69** Serial Transfer (2) Timing



**Figure 2.70** Serial Transfer (Receive) Sample Program

**Caution:** In this example, if there is a rising edge (B) of the transfer clock between the instruction following SELF0 and the SELF1 instruction, an error will occur. The transfer clock should be set with a sufficiently long cycle in relation to the cycle clock.

- Set SCKn to "0" 1 Tcyc before the start of transfer. If less than 1 Tcyc, correct data will not be obtained.
- To set Pn to "output", PnFCR must be set to "0" before PnDDR. If PnDDR is set first, "0" may be output from Pn when PnDDR is set. This applies both to SIO0 and SIO1.

# Dot Matrix LCD Controller

The LCD controller/driver automatically reads data stored in display RAM and generates the signals to drive the dot matrix LCD. The display mode is a graphics mode in which one bit of data in display RAM corresponds to on/off of one dot on the LCD.

The dot matrix LCD controller/driver consists of the following circuit blocks.

- Display RAM (XRAM)
- Display control register
- LCD power supply

## Functions

- Display duty cycle: 1/33
- Display bias: 1/5
- Graphics display
- LCD instruction display on/off
- Graphics display capability 48 (horizontal) x 32 (vertical) matrix + 4 mode icons

The following Special Function Registers must be operated to control the display.

- MCR: display on/off control
- STAD: display start address control
- CNR: horizontal byte number control
- TDR: display duty cycle control
- VCCR: display contrast control
- XBNK: display RAM bank address control

## Display RAM (XRAM)

The display RAM consists of two banks of 96 x 8 bits for dot matrix control and three 6-bit banks for icon control.

The LCD controller/driver reads the data stored in XRAM and generates the signal to drive the LCD.

---

**Caution:** Before reading from or writing to XRAM, set the system clock to the RC oscillator.

---



					Not available for use		
Bank 0	180H	181H	_____	18BH	18CH	-	18FH
	190H	191H	_____	19BH	19CH	-	19FH
			_____			-	
Bank 1	1F0H	1F1H	_____	1FBH	1FCH	-	1FFH
	180H	181H	_____	18BH	18CH	-	18FH
			_____			-	
Bank 2	1F0H	1F1H	_____	1FBH	1FCH	-	1FFH
	180H	181H	_____	18BH	18CH	-	18FH

**Figure 2.71** Display XRAM Configuration

## Display Control Registers

Mode control register (MCR)

Controls display controller operation start/stop, cursor display, and LCD clock division ratio.

**Caution:** The mode control register is write-only. When a bit operation instruction or the INC, DEC, or DBNZ instruction is used on a write-only register, a bit other than the specified bit will be set. Use the following instructions for manipulating this register.  
 MOV, MOV@, ST, ST@, POP  
 When accessing the register, bits 7 - 5 and bit 0 must be set to their fixed values.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
MCR	120H	W	MCR7	MCR6	MCR5	MCR4	MCR3	-	-	MCR0
Reset			0	0	0	0	0	0	0	0

Bit name	Function			
MCR7 (bit 7)	LCD clock division ratio selection			
MCR6 (bit 6)	MCR7	MCR6	MCR5	Division ratio
MCR5 (bit 5)	0	0	0	1/1 * Always set MCR7 - MCR5 to 0
MCR4 (bit 4)	LCD clock 1/2 division ratio select circuit			
	0: Signal determined by MCR7 - MCR5 is divided by 2 and selected as LCD clock 1: Signal determined by MCR7 - MCR5 is selected as LCD clock (direct mode)			
MCR3 (bit 3)	LCD controller control			
	0: LCD controller stop 1: LCD controller start/continue			
MCR0 (bit 0)	Display mode select			
	1: Graphic mode * Always set MCR4 to 1			

### **MCR7, MCR5 (bits 7, 5): LCD clock division ratio**

Be sure to reset MCR7, MCR5 to "0".

### **MCR4 (bit 4): LCD clock 1/2 division select**

This bit controls whether to divide the LCD clock selected with MCR7 - MCR5 by 2.

When reset to "0", the LCD clock is divided by 2.

When set to "0", the LCD clock is not divided.

The frame frequency is as follows.

1/2 division (MCR4 = 0): 82.7 Hz

1/1 division (MCR4 = 1): 165.5 Hz

### **MCR3 (bit 3): LCD controller control**

This bit controls display controller operation start/stop.

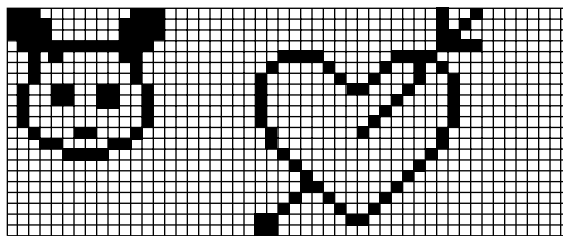
When set to "0", the LCD controller operation starts.

When reset to "0", the LCD controller operation stops. This means that the display will not be updated also when the XRAM contents change. The actual LCD is not switched on and off.

### **MCR0 (bit 0): display mode select**

The display mode should always be set to graphics mode.

Graphics display MCR0 = 1



**Figure 2.72** Dot Matrix Display

**Display start address control register (STAD)**

Controls the display start address.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STAD	122H	R/W	STAD7	STAD6	STAD5	STAD4	STAD3	STAD2	STAD1	STAD0
Reset			0	0	0	0	0	0	0	0

Bit name	Function								
STAD7 (bit 7)	Display RAM start address setting								
STAD0 (bit 0)	STAD7	STAD6	STAD5	STAD4	STAD3	STAD2	STAD1	STAD0	Start address
	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0
	↓	↓							↓
	1	1	1	1	1	1	1	1	255

**Caution:** Changing the STAD value affects not only XRAM bank 0 but also banks 1 and 2, which can cause icons to flash or all icons to be shown simultaneously. In such a case, change the data in bytes 1 to 6 of the display start address specified by STAD as follows.  
 This will cause the game icon only to be shown. Access to bank 2 of XRAM is normally prohibited, but if STAD is set to a value other than 00H, it is allowed.  
 When flash memory access is carried out while STAD is set to a value other than 00H, a part of the screen is rewritten. This is because the BIOS causes the flash memory access icon to be shown.  
 Therefore, STAD should be reset to 00H before calling the BIOS to access the flash memory.

**STAD7, STAD0 (bits 7, 0): XRAM display start address setting**

These bits set the starting address of the display data for the LCD (XRAM 180H is assumed as STAD = 00H).

## Visual Memory Unit (VMU) Hardware Manual

The data changes in 2-byte units.

Start address	XRAM address	STAD7	STAD6	STAD5	STAD4	STAD3	STAD2	STAD1	STAD0
0H	180H (bank 0)	0	0	0	0	0	0	0	0
1H	182H (bank 0)	0	0	0	0	0	0	0	1
2H	184H (bank 0)	0	0	0	0	0	0	1	0
3H	186H (bank 0)	0	0	0	0	0	0	1	1
4H	188H (bank 0)	0	0	0	0	0	1	0	0
5H	18AH (bank 0)	0	0	0	0	0	1	0	1
6H	Not available	0	0	0	0	0	1	1	0
7H	Not available	0	0	0	0	0	1	1	1
8H	190H (bank 0)	0	0	0	0	1	0	0	0
9H	192H (bank 0)	0	0	0	0	1	0	0	1
0AH	194H (bank 0)	0	0	0	0	1	0	1	0
0BH	196H (bank 0)	0	0	0	0	1	0	1	1
0CH	198H (bank 0)	0	0	0	0	1	1	0	0
0DH	19AH (bank 0)	0	0	0	0	1	1	0	1
0EH	Not available	0	0	0	0	1	1	1	0
0FH	Not available	0	0	0	0	1	1	1	1
10H	1A0H (bank 0)	0	0	0	1	0	0	0	0
11H	1A2H (bank 0)	0	0	0	1	0	0	0	1
:	:	:	:	:	:	:	:	:	:
3DH	1FAH (bank 0)	0	0	1	1	1	1	0	1
3EH	Not available	0	0	1	1	1	1	1	0
3FH	Not available	0	0	1	1	1	1	1	1
40H	180H (bank 1)	0	1	0	0	0	0	0	0
41H	182H (bank 1)	0	1	0	0	0	0	0	1
:	:	:	:	:	:	:	:	:	:
7DH	1FAH (bank 1)	0	1	1	1	1	1	0	1
7EH	Not available	0	1	1	1	1	1	1	0
7FH	Not available	0	1	1	1	1	1	1	1
80H	180H (bank 2)	1	0	0	0	0	0	0	0
81H	182H (bank 2)	1	0	0	0	0	0	0	1
82H	184H (bank 2)	1	0	0	0	0	0	1	0
83H - FFH	Not available								

**Caution:** As indicated in the table, some start addresses can lead to operation errors. Do not use xx6H, xx7H, xxEH, xxFH as start addresses.

### Character number register (CNR) 123H

This register may not be accessed by applications.

### Time division register (TDR) 124H

This register may not be accessed by applications.

### Bank address register (XBNK)

Switches the XRAM bank.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
XBNK	125H	R/W	-	-	-	-	-	-	XBNK1	XBNK0
Reset			0	0	0	0	0	0	0	0

Bit name	Function		
XRBK1 (bit 1)	Set display RAM start address		
 XRBK0 (bit 0)	XRBK1	XRBK0	Bank address
	0	0	0
	0	1	1
	1	0	2
	1	1	Not available

### XBNK1, XBNK0 (bits 1, 0): display RAM bank address control

Switches the XRAM bank.

The dot matrix display RAM banks 0 and 1 have a capacity of 96 bytes each. Applications can access only banks 0 and 1.

XRAM bank 2 contains 6 bytes and serves for the icons that indicate the VMU operation mode.

**Caution:** Applications may not manipulate XRAM bank 2.

### LCD contrast control register (VCCR)

This register controls the LCD on/off state.

**Caution:**

- The unit does not incorporate a contrast control.
- The LCD contrast control register is write-only. When a bit operation instruction or the INC, DEC, or DBNZ instruction is used on a write-only register, a bit other than the specified bit will be set. Use the following instructions for manipulating this register.  
MOV, MOV@, ST, ST@, POP

When accessing the register, bits 5 to 0 must be set to their fixed values.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
VCCR	127H	W	VCCR7	VCCR6	VCCR5	VCCR4	VCCR3	VCCR2	VCCR1	VCCR0
Reset			0	0	0	0	0	0	0	0

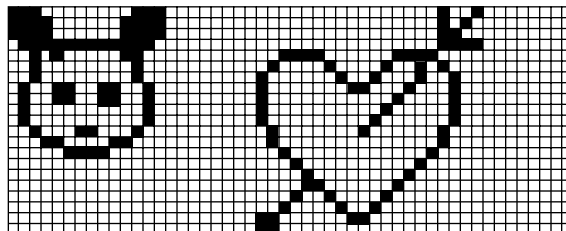
Bit name	Function
VCCR7 (bit 7)	Liquid crystal display control 0: Liquid crystal display OFF 1: Liquid crystal display ON
VCCR6 (bit 6)	LCD RAM access control 0: CPU RAM access enabled 1: CPU RAM access disabled
VCCR5 (bit 5)   VCCR0 (bit 0)	* Always set VCCR5 - VCCR0 to 0

### VCCR7 (bit 7): LCD display control

This bit specifies whether display is carried out or not.

When reset to "0", power to the LCD is shut off, so that the display is deactivated.

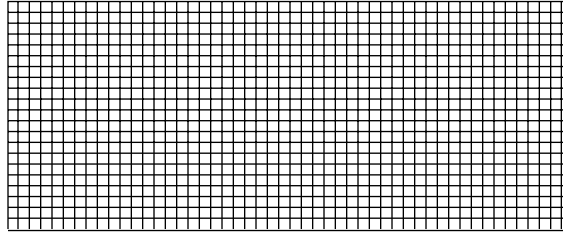
(1) Liquid crystal display ON (VCCR7 = 1)



**Figure 2.73** LCD ON State

When set to "0", power to the LCD is supplied, so that the display is activated.

(2) Liquid crystal display OFF (VCCR7 = 0)



**Figure 2.74** LCD OFF State

---

**Caution:** Always start the display controller (MCR3 = 1) before activating the display (VCCR7 = 1). To deactivate the display, first set VCCR7 to "0" and then set MCR3 to "0".

---

### VCCR6 (bit 6): LCD display RAM access control

When the quartz oscillator is used as system clock and LCD display is on, be sure to disable access from the CPU to the XRAM (VCCR6 = 1) after changing the system clock.

When reading from or writing to XRAM, or when the RC oscillator is used for the system clock and the display is used, enable access from the CPU to the XRAM (VCCR6 = 0).

The procedure for changing the system clock while the display is used is as follows.

- RC oscillator → quartz oscillator  
VCRR6 = 1  
OCR5 = 1, OCR4 = 0
- Quartz oscillator → RC oscillator  
VCRR6 = 0  
OCR5 = 0, OCR4 = 0 (RC oscillator)

### VCCR5 - VCCR0 (bits 5 - 0)

Always set these bits to "0".

---

**Caution:** When using the LCD, set the VCCR last.

---

## External Interrupt Function

The VMU custom chip has a function that detects external input signals on P70/INT0, P71/INT1, P72/INT2/TOIN, and P73/INT3/TOIN and generates interrupt requests to four vector addresses.

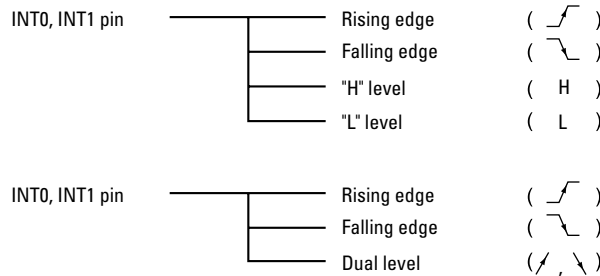
The signal types to be detected can be selected by the application. P70 is used for detecting when the VMU is connected to the controller. P71 is used for low-voltage detection.

### Detection pins and interrupt vectors

Pin	Vector address	Pin	Vector address
P70/INT0	003H	P72/INT2/TOIN Pin	013H
P71/INT1	00BH	P73/INT3/TOIN Pin	01BH

### Signals that can be detected

The priority ranking of the INT0 and INT1 pin interrupts can be set to either "High" or "Low" by the master interrupt enable control register (IE). When set to "High", interrupt processing is carried out regardless of the master interrupt enable setting. The priority ranking of interrupts other than INT0 and INT1 can be set to either "High" or "Low" by the interrupt priority control register (IP). A noise filter with switchable time constant is connected to the P73/INT3/TOIN pin.



**Figure 2.75** *Interrupt Detection Signals*

### Detection of other VMU unit

When another VMU unit is connected, the values at pins P70 through P73 are as follows.

P70	P72	P73	
Connected to VMU	L	L	H
Not connected to VMU	L	L	L

To use the external interrupt function, the following Special Function Registers must be operated. I01CR, I23CR, ISL, IE



## Circuit Configuration

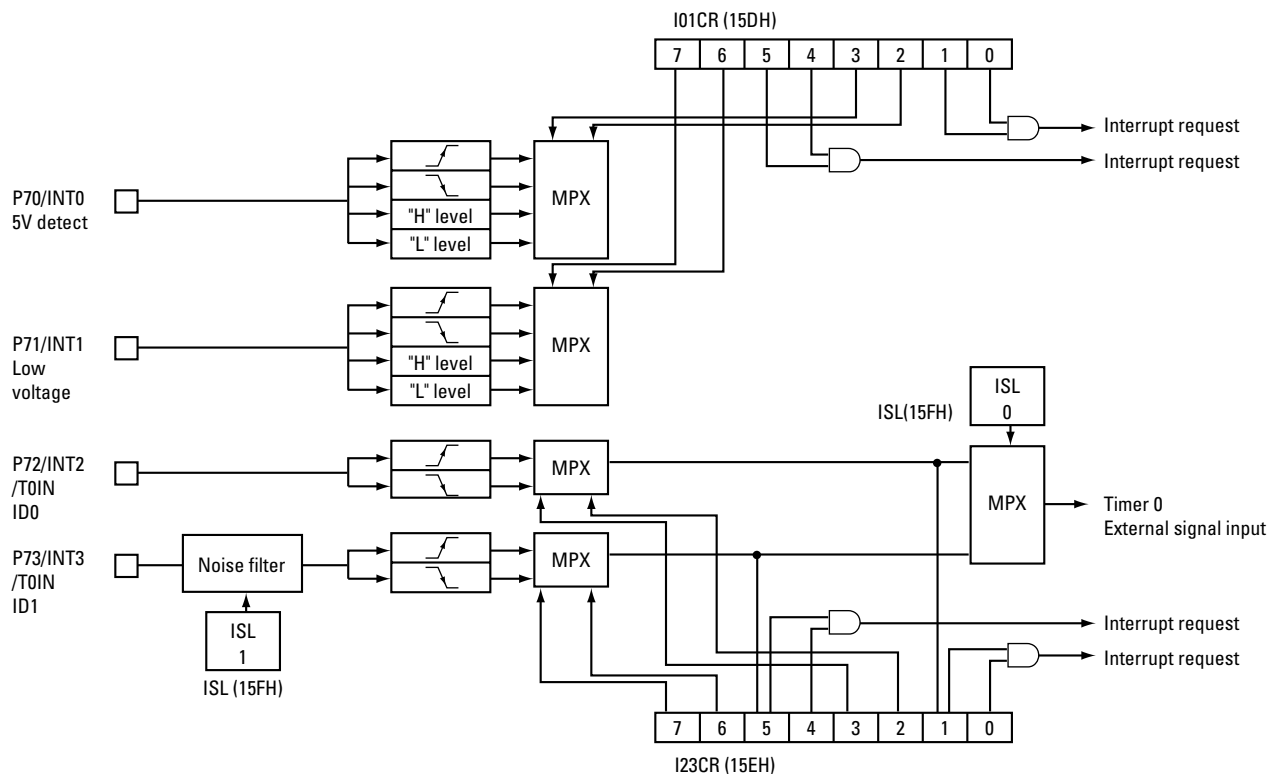


Figure 2.76 External Interrupt Circuit Block Diagram

## Related Registers

External interrupt 0, 1 control register (I01CR)

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I01CR	15DH	R/W	I01CR7	I01CR6	I01CR5	I01CR4	I01CR3	I01CR2	I01CR1	I01CR0
Reset			0	0	0	0	0	0	0	0

Bit name	Function		
I01CR7 (bit 7) I01CR6 (bit 6)	INT1 detection level/edge select		
	I01CR7	I01CR6	INT1 interrupt condition
	0	1	Detect falling edge
	0	1	Detect "L" level
	1	0	Detect falling edge
	1	1	Detect "H" level
I01CR5 (bit 5)	INT1 interrupt source		
	0: Interrupt source disabled 1: Interrupt source enabled		
I01CR4 (bit 4)	INT1 interrupt control		
	0: Interrupt disabled 1: Interrupt enabled		
I01CR3 (bit 3) I01CR2 (bit 2)	INT0 detection level/edge select		
	I01CR3	I01CR2	INT0 interrupt condition
	0	0	Detect falling edge
	0	1	Detect "L" level
	1	0	Detect falling edge
	1	1	Detect "H" level
I01CR (bit 0)	INT0 interrupt source		
	0: Interrupt source disabled 1: Interrupt source enabled		
I01CR1 (bit 1)	INT0 interrupt control		
	0: Interrupt disabled 1: Interrupt enabled		

### I01CR7, I01CR6 (bits 7, 6): INT1 detection level/edge select

Selects the INT1 interrupt condition for signals input on the P71/INT1 pin.

I01CR7	I01CR6	INT1 interrupt condition
0	0	Detect falling edge
0	1	Detect "L" level
1	0	Detect rising edge
1	1	Detect "H" level

**Note:** When level detection is used, an interrupt is generated continuously while the signal is at "High" or "Low" level.

---

### I01CR5 (bit 5): INT1 interrupt source

This bit is set if the condition specified by bits I01CR7 and I01CR6 is met. If INT1 interrupt is enabled (I01CR4 = 1), the interrupt vector 00BH is called and interrupt processing begins.

---

**Caution:** This flag is not reset automatically. It must be reset by the application.

---

### I01CR4 (bit 4): INT1 interrupt enable control

This bit enables or disables the external INT1 interrupt.

When set to "0", INT1 interrupt processing is carried out if I01CR5 is set.

When reset to "0", interrupt processing is not carried out.

### I01CR3, I01CR2 (bits 3, 2): INT0 detection level/edge select

Selects the INT0 interrupt condition for signals input on the P70/INT0 pin.

I01CR3	I01CR2	INT0 interrupt condition
0	0	Detect falling edge
0	1	Detect "L" level
1	0	Detect rising edge
1	1	Detect "H" level

**Note:** When level detection is used, an interrupt is generated continuously while the signal is at "High" or "Low" level.

---

### I01CR1 (bit 1): INT0 interrupt source

This bit is set if the condition specified by bits I01CR3 and I01CR2 is met. If INT0 interrupt is enabled (I01CR0 = 1), the interrupt vector 0003H is called and interrupt processing begins.

---

**Caution:** This flag is not reset automatically. It must be reset by the application.

---

### I01CR0 (bit 0): INT0 interrupt enable control

This bit enables or disables the external INT0 interrupt.

When set to "0", INT0 interrupt processing is carried out if I01CR1 is set.

When reset to "0", interrupt processing is not carried out.

---

## External interrupt 2, 3 control register (I23CR)

For details, refer to “Timer/Counter 0 (T0)”, section “External Interrupt 2, 3 Control Register (I23CR)”.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I23CR	15EH	R/W	I23CR7	I23CR6	I23CR5	I23CR4	I23CR3	I23CR2	I23CR1	I23CR0
Reset			0	0	0	0	0	0	0	0

Bit name	Function
I23CR7 (bit 7)	INT3 rising edge detect control
	0: Detect disabled 1: Detect enabled
I23CR6 (bit 6)	INT3 falling edge detect control
	0: Detect disabled 1: Detect enabled
I23CR5 (bit 5)	INT3 interrupt source
	0: Interrupt source disabled 1: Interrupt source enabled
I23CR4 (bit 4)	INT3 interrupt control
	0: Interrupt disabled 1: Interrupt enabled
I23CR3 (bit 3)	INT2 rising edge detect control
	0: Detect disabled 1: Detect enabled
I23CR2 (bit 2)	INT2 falling edge detect control
	0: Detect disabled 1: Detect enabled
I23CR1 (bit 1)	INT2 interrupt source
	0: Interrupt source disabled 1: Interrupt source enabled
I23CR0 (bit 0)	INT2 interrupt control
	0: Interrupt disabled 1: Interrupt enabled

## Input Signal Select Register (ISL)

For details, refer to “Timer/Counter 0 (T0)”, section “External Signal Select Register (ISL)”.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ISL	15FH	R/W	-	-	ISL5	ISL4	ISL3	ISL2	ISL1	ISL0
Reset			H	H	0	0	0	0	0	0

Bit name	Function		
ISL5 (bit 5) ISL4 (bit 4)	Base timer clock select		
	ISL5	ISL4	
	1	1	Timer/counter T0 prescaler
	0	1	Cycle clock
	X	0	Quartz oscillator
ISL3 (bit 3)	Use prohibited		
	0: fBST/16 (fixed) 1: Not allowed		
ISL2 (bit 2) ISL1 (bit 1)	Noise filter time constant select		
	ISL2	ISL1	Time constant
	1	1	16Tcyc
	0	1	64Tcyc
	X	0	1Tcyc
ISL0 (bit 0)	T0 clock input pin select		
	0: P72/INT2/TOIN pin 1: P73/INT3/TOIN pin		

### Master interrupt enable register (IE)

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
IE	108H	R/W	IE7	-	-	-	-	-	IE1	IE0
Reset			0	H	H	H	H	H	0	0

Bit name	Function			
IE7 (bit 7)	Master interrupt control (high level, low level)			
	0: All interrupt requests disabled 1: All interrupt requests enabled			
IE1 (bit 1) IE0 (bit 0)	INT0, INT1 interrupt priority control			
	IE1	IE0	INT1 priority level	INT0 priority level
	0	0	Highest	Highest
	1	0	Low	Highest
X	1	Low	Low	

### IE7 (bit 7): master interrupt enable control

Enables or disables acceptance of all interrupts, regardless of priority level.

When set to "1", all interrupt requests are enabled.

When reset to "0", interrupts of "high" and "low" priority are disabled.

### IE1 - IE0 (bits 1 - 0): INT0, INT1 interrupt priority control

Controls the priority level of external interrupts.

IE1	IE0	INT1 priority level	INT0 priority level
0	0	Highest	Highest
1	0	Low	Highest
X	1	Low	Low

---

**Caution:**

- INT0 and INT1 can be set to "low" priority but not to "high" priority with IE7.
- It is not possible to set the external interrupt INT1 only to "high" priority.

---

## Port Interrupt Functions

In addition to its digital I/O functions, port 3 can be used to generate an interrupt in response to an external input signal, or to cancel the sleep (HALT) condition.

A port interrupt can be implemented through port 3.

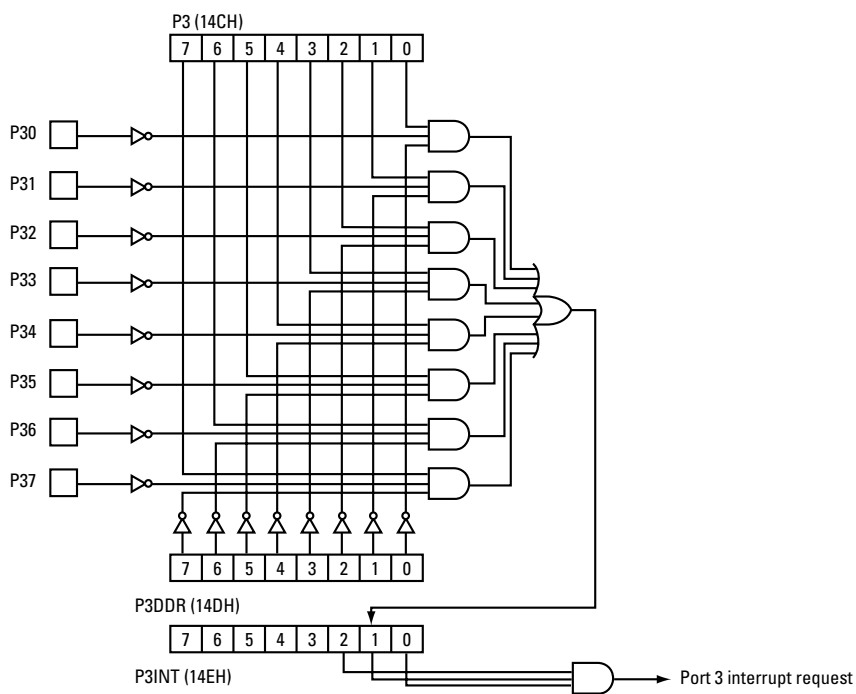
### Function

In addition to its digital I/O functions, port 3 generates an interrupt when it detects a "Low" level signal.

To use the port interrupt function, the following Special Function Registers must be operated.

P3, P3DDR, P3INT, IE

### Circuit Configuration



**Figure 2.77** Port 3 Interrupt Circuit Block Diagram

## Related Registers

Port 3 interrupt control register (P3INT)

For details, refer to the section “Port 3 Interrupt Control Register (P3INT)” in “Port 3”.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
P3INT	14EH	R/W	-	-	-	-	-	P32INT	P31INT	P30INT
Reset			H	H	H	H	H	0	0	0

Bit name	Function
P32INT2 (bit 2)	Port 3 interrupt control flag
	0: Port 3 interrupt generation disabled 1: Port 3 interrupt generation enabled
P32INT1 (bit 1)	Port 3 interrupt source flag
	0: Interrupt source disabled 1: Interrupt source enabled
P32INT0 (bit 0)	Port 3 interrupt request control
	0: Interrupt request disabled 1: Interrupt request enabled

### Master interrupt enable control register (IE)

For details, refer to the section “Master interrupt Control Register (IE)” in “External Interrupt Functions”.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
IE	108H	R/W	IE7	-	-	-	-	-	IE1	IE0
Reset			0	H	H	H	H	H	0	0



Bit name	Function			
IE7 (bit 7)	Master interrupt control (high level, low level)			
	0: All interrupt requests disabled 1: All interrupt requests enabled			
IE1 (bit 1) IE0 (bit 0)	INT0, INT1 interrupt priority control			
	IE1	IE0	INT1 priority level	INT0 priority level
	0	0	Highest	Highest
	1	0	Low	Highest
X	1	Low	Low	

## Operation Description

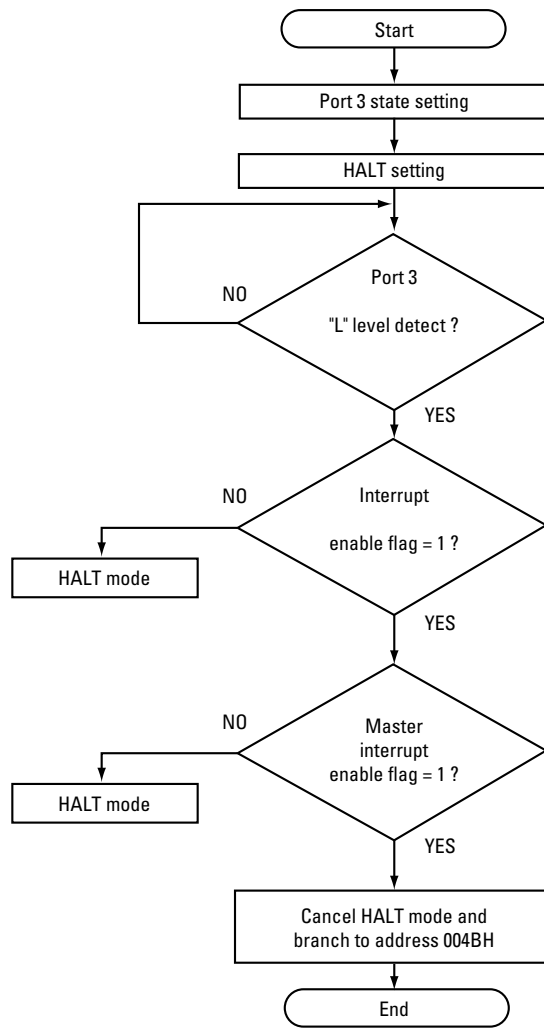
### Port 3 interrupt

1. Set bit 2 of the port 3 control register (P3INT) to "1". This selects port 3 interrupt.
2. Among the pins of port 3 (P37 to P30), select the Special Function Register on which "Low" level detection should occur. The following conditions must be met to accept a port 3 interrupt.
  - The corresponding bit in the port 3 control register (P3DDR) must be set to input mode.  
 $P3mDDR = 0$  ( $m = 0$  to  $7$ )
  - The corresponding bit in the port 3 register (P3) must be set.  
 $P3n = 1$  ( $m = 0$  to  $7$ )
3. When a "Low" level is detected, the interrupt source is set to "1". If the interrupt request enable flag has been set, an interrupt request is generated, and if the master interrupt enable flag has been set, the interrupt vector 004BH is called.
4. If the conditions listed in 2. are met while in HALT mode, the HALT mode is terminated and the interrupt vector 004BH is called.

## State Transition

The flowchart below shows activation and cancellation of HALT mode.

HALT mode state transition



**Figure 2.78** Flow Chart

## VMU Work RAM

The VMU contains 256 bytes x 2 banks of RAM to be used as communications buffer when connected to the Dreamcast. When not connected to the Dreamcast, this RAM is available for applications.

To determine whether data transfer with the Dreamcast is being carried out, check the ASEL flag in the VSEL register. When the flag is "1", data transfer is in progress.

Note that data integrity will not be assured if an application writes to this RAM while data transfer is in progress.

### Work RAM Control Registers

VMU control register (VSEL)

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
VSEL	163H	R/W	-	-	-	INCE	-	-	SIOSEL	ASEL
Reset			H	H	H	0	H	H	0	0

The application can alter only bit 4. Be sure to use a bit level instruction.

#### **INCE (bit 4): VTRBF address counter automatic increment**

This bit controls the automatic incrementing of the address counter when reading/writing VTRBF. When set to "1", the address counter is automatically incremented by 1 after VTRBF has been accessed. When reset to "0", the address counter maintains its setting.

#### **SIOSEL (bit 1): P1 port use select control**

Specifies whether the P1 port (P10 to P15) is to be used as a normal I/O port for synchronous serial communication or as dedicated Dreamcast interface.

#### **ASEL (bit 0): VTRBF address input select control**

Controls access to the VTRBF used as buffer for the VMU and dedicated Dreamcast interface.

**Table 2.27 Work RAM access address (VRMAD1, VRMAD2)**

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
VRMAD1	164H	R/W	VRMAD7	VRMAD6	VRMAD5	VRMAD4	VRMAD3	VRMAD2	VRMAD1	VRMAD0
Reset			0	0	0	0	0	0	0	0

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
VRMAD2	165H	R/W	-	-	-	-	-	-	-	VRMAD8
Reset			H	H	H	H	H	H	H	0

Set the address for reading/writing the work RAM (VTRBF). VRMAD1 specifies the lower 8 bits of the address and VRMAD2 the bank. When bit 4 of VSEL is set to "1", VRMAD is incremented each time the VTRBF is accessed.

**Table 2.28 Work RAM (VTRBF)**

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
VTRBF	166H	R/W	VTRBF7	VTRBF6	VTRBF5	VTRBF4	VTRBF3	VTRBF2	VTRBF1	VTRBF0
Reset			0	0	0	0	0	0	0	0

This register serves for reading and writing data in the address specified by VRMAD.

When this register is written to, the data are written to the RAM address specified by VRMAD.

When this register is read from, the data are read from the RAM address specified by VRMAD.

When bit 4 of VSEL is set to "1", VRMAD is incremented each time the register is accessed.

### Accessing Work RAM

To access work RAM, store the desired RAM address in the VRMAD1 and VRMAD2 registers. Then read or write to VTRBF to access data in the work RAM.

---

**Caution:** The VRMAD1 and VRMAD2 registers are provided with an auto-increment function. To enable this function, set the INCE bit of VSEL to "1". To disable it, set the bit to "0".

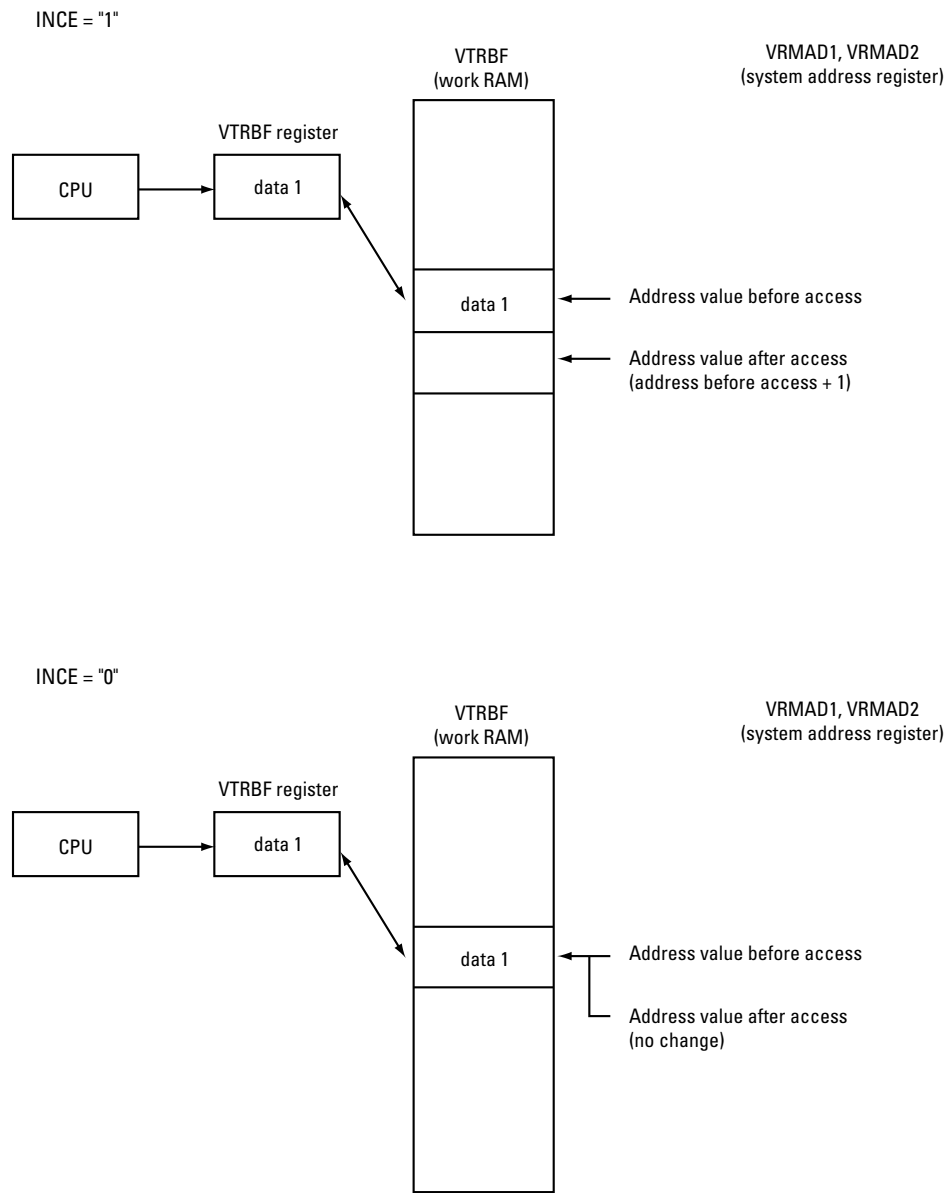
---

### Precautions for Using Work RAM Address Register

The work RAM access procedure is shown in Fig. below.

To access work RAM from an application, specify the work RAM address in the VRMAD1 and VRMAD2 registers.

If the INCE flag of the VSEL register is set to "1", VRMAD is incremented automatically after each time VTRBF is accessed. The program therefore must be written so as to take the status of the INCE flag of the VSEL register into account.



**Figure 2.79** Work RAM Access

# **Flash Memory**

The VMU custom chip incorporates 128 KB of flash memory (EEPROM = Electrically Erasable Programmable ROM) which can be used to store application program code or data.

## **Features and Functions**

- Capacity: 131072 x 8 bits (program/ data area)
- Programmable/erasable in block (page) units  
1 block = 128 bytes (1 page)
- Number of write/erase cycles:  
50,000 times/page (managed by program) (at 25°C ambient temperature)
- Integrated step-up circuit for writing
- Write end detection possible (by calling OS program)  
Toggle bit principle  
Data polling principle
- Software batch erase possible

## **Accessing Program/Data Area of Flash Memory**

The program/ data area of flash memory is accessed by calling an OS program. For details, refer to the "BIOS" section.

When connected to the Dreamcast, an application can be transferred to the VMU from the Dreamcast. For details on the transfer procedure, refer to the section on the `buSaveExecFile()` function in the SEGA Library Manual Vol. 2.

By connecting the development computer to the Dreamcast with a special cable and using the dedicated Memory Card Utility, an application can be transferred to the VMU. For details, refer to the VMU Tutorial.

# ***Control Functions***

---

This section contains information about the interrupt controller and the system clock.

## **Interrupt Functions**

Interrupts are used to temporarily interrupt a running program in order to execute other program with higher priority. The VMU incorporates circuits for generating 13 types of interrupts. These are shown in the table below.

---

**Caution:** Some interrupt processing functions cannot be set freely by applications.

---

## Interrupt Types

**Table 2.29 Interrupt Table**

Priority sequence	Interrupt type	Internal/ External	Vector address	Interrupt request	Source flag	Enable flag	Register address	Priority setting
1	External interrupt INT0	External	0003H	P70/INT0 event detection	I01CR1	I01CR0	15DH	Highest/low
2	External interrupt INT1	External	000BH	P71/INT1 event detection	I01CR5	I01CR4	15DH	
3	External interrupt INT2	External	0013H	P72/INT2 event detection	I23CR1	I23CR0	15EH	High/low
	Timer/counter TOL (lower 8 bits)	Internal		Timer/counter TOL lower 8 bits overflow	TOCNT1	TOCNT0	110H	
4	External interrupt INT3	External	001BH	P73/INT3 event detection	I23CR5	I23CR4	15EH	High/low
	Base timer	Internal		Base timer overflow	BTCR1 BTCR3	BTCR0 BTCR2	17FH	
5	Timer/counter TOH (lower 8 bits)	Internal	0023H	Timer/counter TOL lower 8 bits overflow	TOCNT1	TOCNT0	110H	High/low
6	Timer T1	Internal	002BH	Timer T1L overflow	T1CNT1	T1CNT0	118H	High/low
				Timer T1H overflow	T1CNT3	T1CNT2		
7	SIO0	Internal	0033H	SIO0 end detect	SCON01	SCON00	130H	High/low
8	SIO1	Internal	003BH	SIO1 end detect	SCON11	SCON10	134H	High/low
9	VMU interrupt	Internal	0043H	VMU transfer receive end detect	RFB	RFBENA	160H/161H	High/low
10	Port 3 interrupt (P32INT = 1)	External	004BH	Port 3 "L" level detect	P31INT	P30INT	14EH	High/low

**Caution:** • The priority ranking indicates which interrupt is handled first if several interrupts are generated simultaneously. The priority ranking changes if specified in the interrupt priority control register (IP).



## Interrupt Function Operation

When an interrupt as listed in Table 5-1 is generated, the corresponding interrupt request flag is set. This indicates to the interrupt control circuit that an interrupt request has occurred.

The interrupt control circuit accepts interrupts in the order of their priority. There are three priority levels: "highest", "high", and "low". To enable interrupts with "high" and "low" priority, the master interrupt flag (IE7) must also be set in addition to the individual interrupt enable flags. IE7 controls interrupts with "high" and "low" priority. If INT0 or INT1 are set to "highest" priority by the interrupt priority control flag (IE1, IE0), interrupt processing occurs regardless of the master interrupt enable flag.

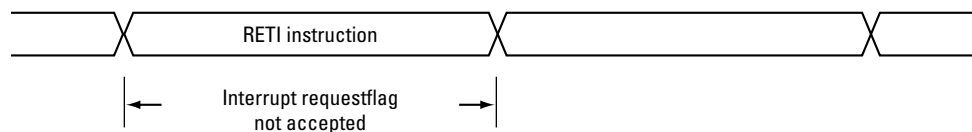
Interrupt sources with a priority ranking from 3 to 9 can be specified as having either "high" or "low" priority by the interrupt priority control register (IP).

When an interrupt is generated, the interrupt control circuit waits until the currently executing instruction is completed. Then it stores the program counter (PC) contents in the stack (in RAM) and executes the interrupt processing routine. This operation uses 2 bytes of stack (RAM) and increments the stack pointer (SP) by +2. After returning from the interrupt processing routine, the stack pointer is decremented by -2.

By executing a RETI instruction at the end of the interrupt processing routine, execution returns to the original program.

Interrupt nesting is possible and can be up to 3 levels deep.

During execution of the RETI instruction or an instruction (MOV, ST, etc.) that writes to one of the special function registers listed below, or while writing to flash memory, interrupt request flag acceptance processing is not performed.



**Figure 2.80** Cycle Without Interrupt Processing

To use the interrupt function, the following Special Function Registers must be operated.

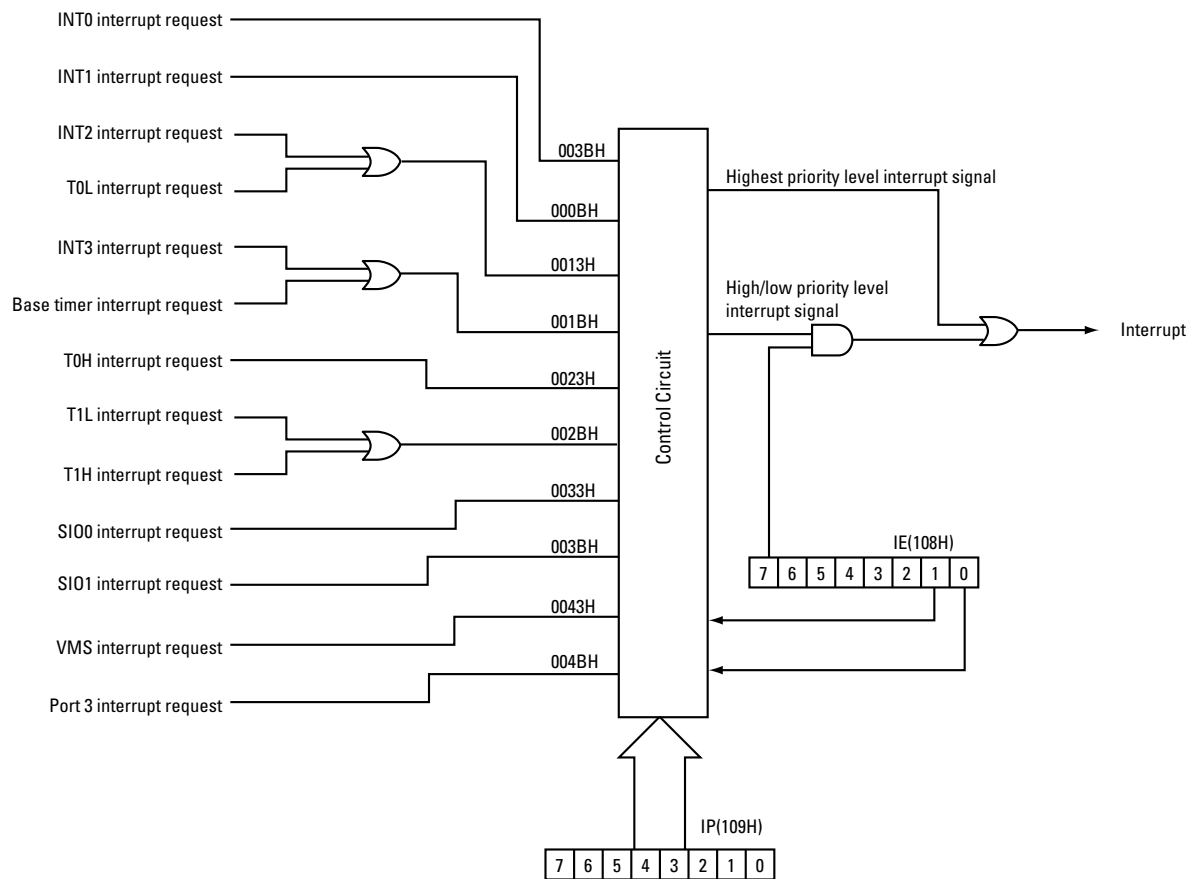
IE, IP, SP < Caution >, special function registers in the function block that accepts the interrupt

---

**Caution:** System program settings are made during a hardware reset. It is not possible to directly manipulate the SP from an application.

---

## Circuit Configuration



**Figure 2.81** *Interrupt Function 1 Block Diagram*

## Related Registers

### Master interrupt enable control register (IE)

For details, refer to the section “Master Interrupt Control Register (IE)” in “External Interrupt Functions”.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
IE	108H	R/W	IE7	-	-	-	-	-	IE1	IE0
Reset			0	H	H	H	H	H	0	0

Bit name	Function			
IE7 (bit 7)	Master interrupt control (high level, low level)			
	0: All interrupt requests disabled 1: All interrupt requests enabled			
IE1 (bit 1) IE0 (bit 0)	INT0, INT1 interrupt priority control			
	IE1	IE0	INT1 priority level	INT0 priority level
	0	0	Highest	Highest
	1	0	Low	Highest
	X	1	Low	Low

## Interrupt Priority Control Register (IP)

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
IP	109H	R/W	IP7	IP6	IP5	IP4	IP3	IP2	IP1	IP0
Reset			0	-	0	0	0	0	0	0

Bit name	Function
IP7 (bit 7)	Port 3 interrupt priority level setting
	0: Low 1: High
IP5 (bit 5)	SIO1 interrupt priority level setting
	0: Low 1: High
IP4 (bit 4)	SIO0 interrupt priority level setting
	0: Low 1: High
IP3 (bit 3)	T1 interrupt priority level setting
	0: Low 1: High
IP2 (bit 2)	TOH interrupt priority level setting
	17 0: Low 1: High
IP1 (bit 1)	INT3 and base timer interrupt priority level setting
	0: Low 1: High
IP0 (bit 0)	INT2 and TOL interrupt priority level setting
	0: Low 1: High

**IP7 (bit 7): port 3 interrupt priority level setting**

This bit selects either "high" (1) or "low" (0) for the port 3 interrupt priority level.

When set to "1", the priority setting for this interrupt is "high", giving the interrupt higher priority than the INT0 and INTO interrupts (IE0 = 1).

When reset to "0", the priority setting for this interrupt is "low".

**IP5 (bit 5): SIO1 interrupt priority level setting**

This bit selects either "high" (1) or "low" (0) for the SIO1 interrupt priority level.

When set to "1", the priority setting for this interrupt is "high", giving the interrupt higher priority than the INT0 and INTO interrupts (IE0 = 1).

When reset to "0", the priority setting for this interrupt is "low".

**IP4 (bit 4): SIO0 interrupt priority level setting**

This bit selects either "high" (1) or "low" (0) for the SIO0 interrupt priority level.

When set to "1", the priority setting for this interrupt is "high", giving the interrupt higher priority than the INT0 and INTO interrupts (IE0 = 1).

When reset to "0", the priority setting for this interrupt is "low".

**IP3 (bit 3): T1 interrupt priority level setting**

This bit selects either "high" (1) or "low" (0) for the T1 interrupt priority level.

When set to "1", the priority setting for this interrupt is "high", giving the interrupt higher priority than the INT0 and INTO interrupts (IE0 = 1).

When reset to "0", the priority setting for this interrupt is "low".

**IP2 (bit 2): T0H interrupt priority level setting**

This bit selects either "high" (1) or "low" (0) for the T0H interrupt priority level.

When set to "1", the priority setting for this interrupt is "high", giving the interrupt higher priority than the INT0 and INTO interrupts (IE0 = 1).

When reset to "0", the priority setting for this interrupt is "low".

**IP1 (bit 1): INT3/base timer interrupt priority level setting**

This bit selects either "high" (1) or "low" (0) for the INT3/base timer interrupt priority level.

When set to "1", the priority setting for this interrupt is "high", giving the interrupt higher priority than the INT0 and INTO interrupts (IE0 = 1).

When reset to "0", the priority setting for this interrupt is "low".

**IP0 (bit 0): INT2/TOL interrupt priority level setting**

This bit selects either "high" (1) or "low" (0) for the INT2/TOL interrupt priority level.

When set to "1", the priority setting for this interrupt is "high", giving the interrupt higher priority than the INT0 and INTO interrupts (IE0 = 1).

When reset to "0", the priority setting for this interrupt is "low".

## Interrupt Priority Ranking

The priority ranking of interrupts is as follows.

Highest level > high level > low level

If multiple interrupts of the same priority level are generated simultaneously, the processing order will be as shown in Table below. The multiple interrupt control circuit controls overlapping interrupts, allowing nesting of "low" level → "high" level → "highest" level interrupts.

### Highest level

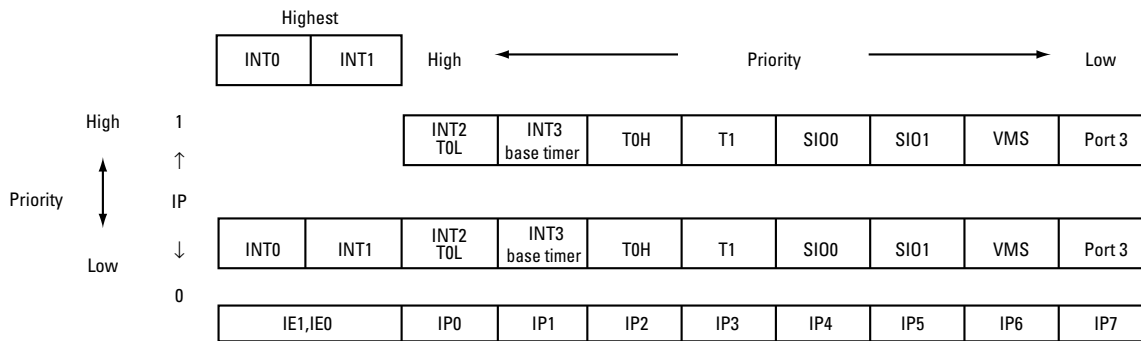
The external interrupts INT0 and INT1 can be set to the "highest" priority level. Interrupts with this priority level are not controlled by the masking enable flag (IE7).

### High level

Interrupt sources other than the external interrupts INT0 and INT1 that correspond to the bits that are set in the interrupt priority control register (IP). Interrupts with this priority level are controlled by the masking enable flag (IE7).

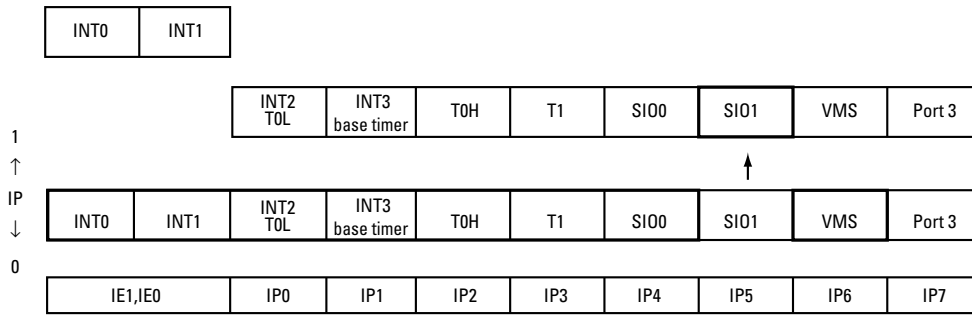
### Low level

Interrupt sources INT0 or INT1 for which "low" level is set in IE0 or IE1 and that correspond to the bits that are reset in the interrupt priority control register (IP). Interrupts with this priority level are controlled by the masking enable flag (IE7).



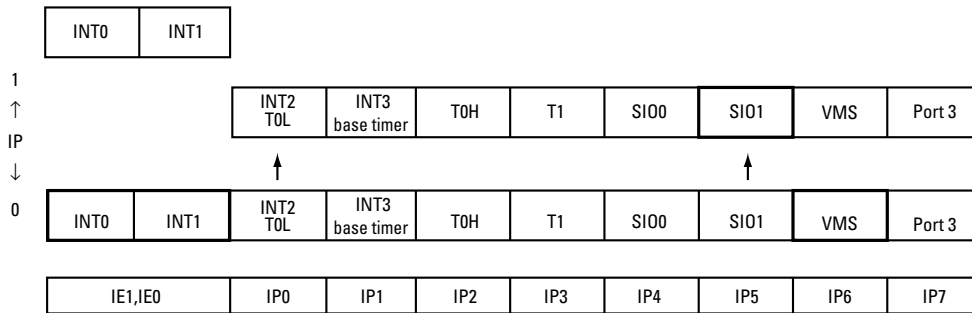
**Figure 2.82** Interrupt Priority Sequence

For example, to give the SIO1 end interrupt higher priority than the INT0 interrupt, set IE0 and IP5 to "1" (IE0 = 1, IP = 00100000B).



**Figure 2.83** SIO1 Interrupt Priority Sequence

To give the SIO1 end interrupt priority between INT2 and INT0, set IE0, IP5, and IP0 to "1" (IE0 = 1, IP = 00100001B).



**Figure 2.84** TOL → SIO1 → INT0 Priority Change

**Multiple interrupt handling**

If a "low" priority interrupt is generated while a "high" priority interrupt routine is being executed, the "low" priority interrupt is accepted after the "high" priority interrupt routine is completed and one instruction was executed.

If an interrupt routine is being executed and an interrupt of the same priority level is generated, the second interrupt request is not accepted.

**System Clock Generation**

The VMU incorporates two oscillator circuits: a RC oscillator and a quartz oscillator. Either of these can be selected to supply the system clock. This selection is performed through software.

The oscillation frequencies and cycle clock data for these circuits are shown below.

Oscillator	Frequency	Cycle clock	Purpose	Characteristics
RC oscillator	879.236KHz	6.284 ms	Flash memory access	Uses this clock for flash memory For write operations, set the division ratio to 1/6.
Quartz oscillator	32.768KHz	183.105 ms	Standalone operation clock	Processing speed is reduced for preserving battery power

**Note:** Because the RC oscillator consumes more battery power, you should normally use the quartz oscillator for the system clock. Due to individual tolerances in CR circuits, correct audio frequency output will not be obtained with PWM. For PWM, use the quartz oscillator. The tolerance range of the RC oscillator is 600 to 1200 kHz. Applications using the oscillator should be designed for a reference frequency of 879.236 kHz.

---



## Features and Functions

- Generation of system clock to be used as reference for instructions
- System clock can be selected through software, using either RC oscillator or quartz oscillator.
- Generation of base timer clock
- RC oscillator can be stopped through software.
- Two system clocks are generated: system clock 1 (S1) for circuit block that operates also in HALT mode, and system clock 2 (S2) for circuit block that stops to operate in HALT mode

To use the system clock, the following Special Function Registers must be operated.

OCR, PCON

### **Circuit Configuration**

The system clock generator configuration is shown in Fig. below.

#### **Quartz oscillator...②**

This quartz oscillator has an oscillation frequency of 32.768 kHz.

#### **RC oscillator...→**

This circuit uses capacitors (C) and resistors (R) to generate a frequency of 879.236 kHz. The tolerance range is 600 - 1200 kHz.

#### **System clock selector...↪**

Bits 4 and 5 of the oscillation control register (OCR) are used to select either the quartz oscillator or the RC oscillator.

#### **System clock generation circuit...③**

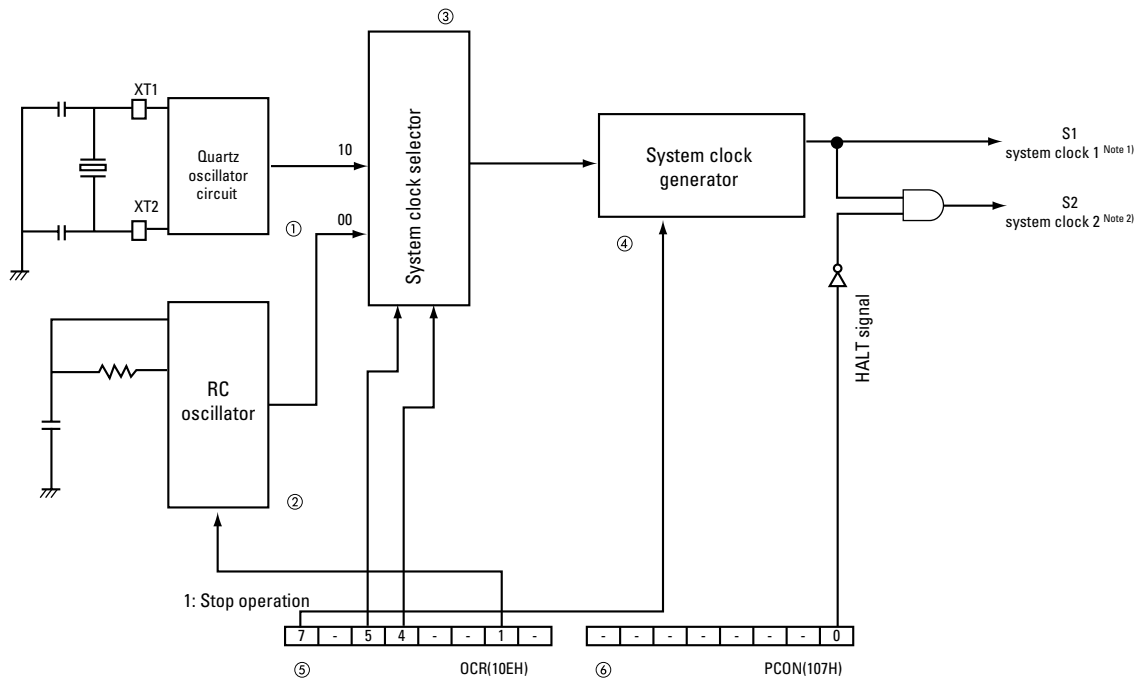
This circuit generates system clock 1 and system clock 2 from the source selected by the system clock selector. System clock 1 (S1) operates while instructions are executed and during HALT mode. System clock 2 (S2) operates while instructions are executed.

#### **Oscillation control register (OCR)... (**

Serves for RC oscillator start/stop, system clock source selection, and cycle clock control.

#### **Power control register (PCON)...**

Sets the standby state (HALT mode).



Note 1): System clock 1 (S1) is used for circuits that operate during instruction execution and in HALT mode.  
 Note 2): System clock 2 (S2) is used for circuits that operate during instruction execution but not in HALT mode.

**Figure 2.85** System Clock Generator Block Diagram

Block status during reset and HALT

**Table 2.30** Operation Status at Standby

Block	Condition	
	Reset	HALT
RC oscillator	Operates	Same as when activated
Quartz oscillator	Stops	Same as when activated
System clock generator	Operates	Operates

**Caution:** Immediately after a hardware reset, the RC oscillator is automatically selected for the system clock. The system BIOS then switches to the quartz oscillator.

## Related Registers

### Oscillation control register (OCR)

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OCR	10EH	R/W	OCR7	-	OCR5	OCR4	-	-	OCR1	-
Reset			0	H	0	0	H	H	0	0

Bit name	Function		
OCR7 (bit 7)	System clock generator control		
	0: Cycle time source is oscillator frequency x 1/2 1: Cycle time source is oscillator frequency x 1/6		
OCR6 (bit 6) OCR4 (bit 4)	System clock select		
	OCR5	OCR4	System clock
	0	0	RC oscillator
	0	1	Prohibited
	1	0	Quartz oscillator
1	1	Prohibited Reset/HALT cancel: RC oscillator	
OCR1 (bit 1)	RC oscillator select		
	0: RC oscillator operation start/continue 1: RC oscillator operation stop		

#### OCR7 (bit 7): system clock generation circuit control

This bit controls whether the clock source for the cycle clock is divided by 12 or by 6.

When set to "1", the cycle clock is 1/6 of the clock source.

When set to "0", the cycle clock is 1/12 of the clock source.

For the VMU, the setting should be as follows.

System clock	OCR7
RC oscillator	OCR7=0/1
Quartz oscillator	OCR7=1

**Caution:** To use the quartz oscillator, be sure to set this bit to "1".  
When using the RC oscillator, select the 1/12 division ratio (OCR7 = "0") except when writing to flash memory.

### OCR5, OCR4 (bit 5, 4): system clock select

This bit selects the system clock. During a hardware reset, the system clock is automatically set to the RC oscillator.

OCR5	OCR4	System clock
0	0	RC oscillator
0	1	Prohibited
1	0	Quartz oscillator
1	1	Prohibited

### OCR1 (bit 1): RC oscillator control

This bit stops/starts the RC oscillator.  
When set to "1", the RC oscillator is stopped.  
When reset to "0", the RC oscillator operates.

**Caution:** Note that negative logic is employed here. "0" means start and "1" means stop.

### Power control register (PCON)

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PCON	107H	R/W	-	-	-	-	-	-	-	PCON0
Reset			H	H	H	H	H	H	0	0

Bit name	Function
PCON0 (bit 0)	HALT mode control
	0: 1: Set to HALT mode

### **PCON0 (bit 0): HALT mode control**

This bit sets the VMU to the sleep state.

When set to "1", the VMU custom chip goes into HALT mode, causing the VMU to enter the sleep state. Program execution stops at the address where HALT was execute, and the oscillator maintains its current state. The system clock 2 (S2) stops.

The HALT mode is canceled by an interrupt. When the HALT mode is canceled, this bit will be automatically reset.

Directly resetting the bit to "0" does not cause a state change.

---

**Note:** In HALT mode, the LCD driver, LCD, timer 0, and timer 1 continue to operate. For details, refer to section on "Sleep Function".

---

## System Clock Operation Mode

There are three types of system clock.

### **RC oscillator**

This clock is selected in the cases listed below. The oscillation frequency of the RC oscillator is 879.236 kHz.

---

**Caution:** Due to the characteristics of RC oscillators, there is a wide variation in frequency. The tolerance range for the VMU is 600 to 1200 kHz.

---

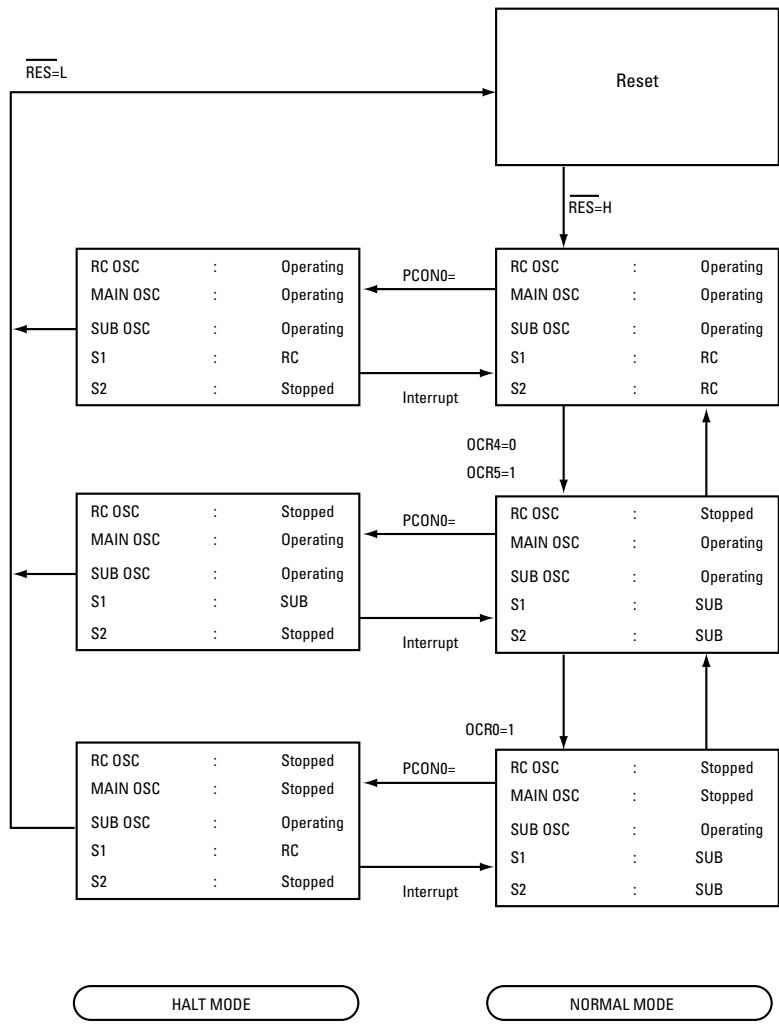
- Hardware reset
- Power-on (battery replacement)

### **Quartz oscillator**

This oscillator allows implementing a slow processing mode with reduced current consumption, for long-term backup. The oscillation frequency is 32.678 kHz.

When the quartz oscillator is used for the system clock, the RC oscillator can be stopped, using the oscillation control register (OCR). This allows a further reduction in current consumption.

The VMU custom chip enters the HALT mode as shown in the state transition diagram of Fig. below.



**Figure 2.86** Clock Operation Mode Transition Diagram

RC OSC:	RC oscillator	SUB:	Quartz oscillator frequency
SUB OSC:	Quartz oscillator circuit control)	PCON0:	power control register bit 0 (HALT
S1:	System clock 1	OCR1:	Oscillation control register bit 1
S2:	System clock 2	OCR4:	Oscillation control register bit 4
RC:	RC oscillator frequency	OCR5:	Oscillation control register bit 5

**Caution:** When switching the system clock to the stopped quartz oscillator, a wait period is required to allow the oscillator to stabilize. For the quartz oscillator in the VMU (32.678 kHz), this wait period is approx. 200 ms.



## **Sleep Function**

The VMU custom chip provides a HALT mode designed to reduce power consumption during program standby and to delay battery exhaustion. In this mode, the CPU does not execute instructions. The sleep mode of the VMU makes use of the HALT mode.

## Related Registers

Power control register (PCON)

For details, refer to the section “Power control register (PCON)” in “System Clock Generation”.

Symbol	Address	R/W	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PCON	107H	R/W	-	-	-	-	-	-	-	PCON0
Reset			H	H	H	H	H	H	0	0

Bit name	Function
PCON0 (bit 0)	HALT mode control
	0: 1: Set to HALT mode

## Standby Operation Status

**Table 2.31 Block Operation Status in Standby Mode**

Item		HALT mode
Setting method		PCON0=1
Oscillation circuits	CF oscillator	Operation continues
	RC oscillator	
	Quartz oscillator	Operation continues
Internal clock	S1	Operation continues
	S2	Operation stops
CPU		Operation stops
I/O ports		Hold data from immediately before HALT mode
RAM		Hold data from immediately before HALT mode
Base timer		Operation continues
Timer 0		Operation continues
Timer 1		Operation continues
Serial transfer		Operation continues
Interrupt circuits		Operation continues
LCD controller		Operation continues
Remote control circuit		Operation continues
Watchdog timer		Operation continues or stops
Released by		Reset Accepted interrupt request

**Note:** If the quartz oscillator is selected for the system clock, stop the RC oscillator through software (OCR1 = 1).

## HALT Mode

The HALT mode allows stopping program execution while keeping the quartz oscillator and RC oscillator circuits running.

Power consumption can be reduced through intermittent operation of the system by recurringly setting HALT mode and having it released in response to an interrupt.

### Setting HALT mode

HALT mode is set by setting bit 0 of the power control register (PCON0).

### Releasing HALT mode

HALT mode can be released in one of two ways: through a hardware reset or through receiving an interrupt request.

#### Releasing HALT mode through hardware reset

When a "Low" level signal is input to the pin, HALT mode is released and the CPU enters the reset state. Returning the pin to "high" level triggers a normal cold start procedure, with the system program executing the VMU initialization routine.

#### Releasing HALT mode through interrupt request

When the master interrupt enable flag (IE7) and interrupt request enable flag are both set and an interrupt request is generated, the HALT mode is canceled. Subsequently, the processing routine corresponding to the interrupt is called.

If HALT mode was activated in interrupt processing routine A, and the interrupt requested generated while in HALT mode has the same or a lower priority level than interrupt A, the interrupt is not accepted and HALT mode will not be canceled.

---

**Caution:**

- If the external interrupts INT0 and INT1 are set to the "highest" priority level, the master interrupt enable flag has no effect.
- Set the interrupt used to cancel HALT mode to a higher priority level than the interrupt in effect when the system entered the HALT mode.

---

**Table 2.32 HALT Mode Cancel Interrupt Priority Levels**

Interrupt level in HALT mode	Interrupt level for HALT mode cancel
Normal level	Low, High, or Highest
Low	High or Highest
High	Highest
Highest	(Cancel by interrupt not possible)

Normal level: No interrupt is present.

## **Hardware Reset Function**

The hardware reset function serves for initializing the VMU for example when the batteries are replaced or while the unit is running.

## External Reset Pin Function

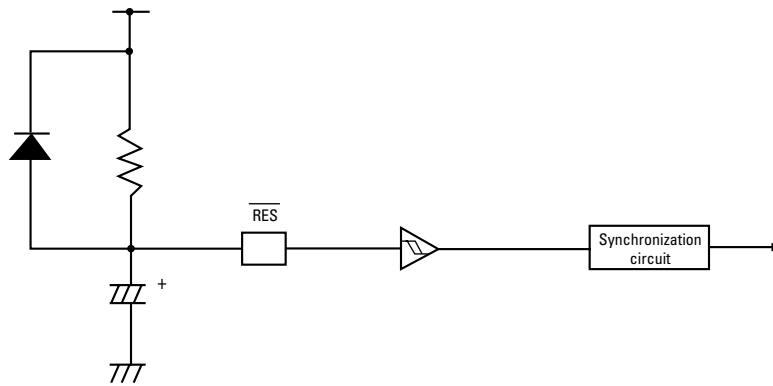
Applying an "L" level signal to the pin for 200 us or more reliably triggers a reset.

---

**Caution:** A very narrow "Low" level pulse can also cause a reset.

---

The configuration of the reset circuit is shown in Fig. below.



**Figure 2.87** Reset Circuit Block Diagram

## Hardware Status During a Reset

When a reset is generated through the pin, the entire hardware is initialized according to the reset signal, which is synchronized with the system clock.

When a reset occurs, the system clock is switched to the RC oscillator. Therefore the hardware is initialized immediately after power-on.

During reset, the program counter (PC) is set to 0000H. The special function registers (SFR) are set to the initial values listed in the Table below.

The contents of RAM, work RAM, stack pointer, and XRAM are maintained.

---

**Caution:** The initial values listed below are the values established by the BIOS after a reset.

---

**Table 2.33 Special Function Register Initial Values**

Symbol	Address	R/W	Designation	Initial value	See page
RAM (bank 0)	000H-0FFH	R/W	Data memory	XXXXXXXX (retained after a reset)	43
RAM (bank 1)	000H-0FFH	R/W	Data memory	XXXXXXXX (retained after a reset)	43
ACC	100H	R/W	Accumulator	00000000	50
PSW	101H	R/W	Program status word	00H00000	52
B	102H	R/W	B register	00000000	51
C	103H	R/W	C register	00000000	51
TRL	104H	R/W	Table reference register lower byte	00000000	54
TRH	105H	R/W	Table reference register upper byte	00000000	54
SP	106H	R/W	Stack pointer	XXXXXXXX	53
PCON	107H	R/W	Power control register	HHHHHH00	158
IE	108H	R/W	Master interrupt enable control register	0HHHHH00	138
IP	109H	R/W	Interrupt priority control register	00000000	151
EXT	10DH	R/W	External memory control register	HHHH0000	-
OCR	10EH	R/W	Oscillation control register	0H00HH00	156
TOCNT	110H	R/W	Timer 0 control register	00000000	67
TOPRR	111H	R/W	Timer 0 prescaler data	00000000	71
TOL	112H	R	Timer 0 low	00000000	71
TOLR	113H	R/W	Timer 0 low reload register	00000000	71
TOH	114H	R	Timer 0 high	00000000	72
TOHR	115H	R/W	Timer 0 high reload register	00000000	72
T1CNT	118H	R/W	Timer 1 control register	00000000	83
T1LC	11AH	R/W	Timer 1 low compare data	00000000	86
T1L	11BH	R	Timer 1 low	00000000	85
T1LR		W	Timer 1 low reload data	00000000	85
T1HC	11CH	R/W	Timer 1 high compare data	00000000	87
T1H	11DH	R	Timer 1 high	00000000	86



T1HR		W	Timer 1 high reload data	00000000	86
MCR	120H	W	Mode control register	00000000	127
STAD	122H	R/W	Start address register	00000000	129
CNR	123H	W	Character count register	H0000000	130
TDR	124H	W	Time division register	HH000000	130
XBNK	125H	R/W	Bank address register	HHHHHH00	130
VCCR	127H	W	LCD contrast control register	00000000	131
SCON0	130H	R/W	SIO0 control register	00H00000	108
SBUF0	131H	R/W	SIO0 buffer	00000000	113
SBR	132H	R/W	SIO baud rate generator	00000000	113
SCON1	134H	R/W	SIO1 control register	00000000	111
SBUF1	135H	R/W	SIO1 buffer	00000000	113
P1	144H	R/W	Port 1 latch	00000000	58
P1DDR	145H	W	Port 1 data direction register	00000000	58
P1FCR	146H	W	Port 1 function control register	10111111	59
P3DDR	14DH	W	Port 3 data direction register	00000000	62
P3INT	14EH	R/W	Port 3 interrupt function control register	11111101	62
P7	15CH	R	Port 7 latch	HHHHXXXX	64
I01CR	15DH	R/W	External interrupt 0, 1 control	00000000	135
I23CR	15EH	R/W	External interrupt 2, 3 control	00000000	137
ISL	15FH	R/W	Input signal select	11000000	138
VSEL	163H	R/W	Control register	11111100	143
VRMAD1	164H	R/W	System address register 1	00000000	144
VRMAD2	165H	R/W	System address register 2	HHHHHHH0	144
VTRBF	166H	R/W	Send/receive buffer	XXXXXXXX	144
BTCCR	17FH	R/W	Base timer control	01000001	101
RAM (XRAM) (Bank 0)	180H-1FBH	R/W	LCD display memory	XXXXXXXX (retained after a reset)	126
RAM (XRAM) (Bank 1)	180H-1FBH	R/W			
RAM (XRAM) (Bank 2)	180H-185H	R/W			



# Programs in ROM

The ROM of the VMU contains the following programs. Together, these are called the system BIOS.

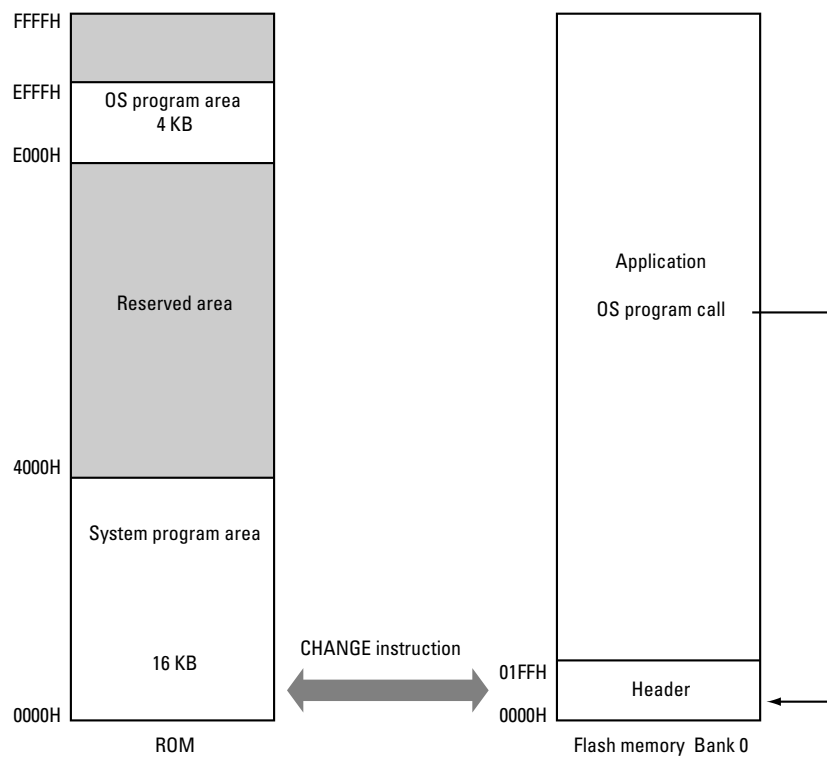


Figure 2.88 ROM Memory Map

## **System Programs**

Programs designed for performing VMU file management, clock display, and control functions when connected to the Dreamcast are called system programs.

These programs are permanently placed in ROM. In addition, the ROM also contains the VMU initialization routine that is executed when a hardware reset is performed.

## **OS Programs**

OS programs perform basic VMU functions such as reading from and writing to flash memory, getting clock data, checking for the low-voltage condition, etc.

These programs correspond to the BIOS in a conventional computer. The programs can be called by an application. Because software interrupts cannot be used, headers defined in the upper region of the flash memory are used to call the programs.

### **Headers**

The assembler file `GHEAD.ASM` supplied with the VMU SDK contains the headers.

By using “include” when compiling an application, the headers are placed in the 0000H - 01FFH range of the flash memory. The size can be changed by changing `GHEAD.ASM`.

The header area defines the interface for switching between a game application and the system application, the interface for calling an OS program from an application, as well as application-specific interrupt vectors.

---

**Caution:** It is not possible to obtain the system BIOS version or VMU hardware revision from an application.

---

# *Memory Space*

---

The system BIOS uses the following memory areas.

## **RAM**

The system BIOS uses the RAM bank 0 for processing. The range from 080H to 0FFH of RAM bank 0 is used as stack area.

RAM bank 0 can generally not be accessed by an application, except for reading the internal clock and the low-voltage auto detect flag.

The stack area (080H to 0FFH of RAM bank 0) can also be accessed by an application, but care must be taken not to corrupt the stack.

The 256 bytes of RAM bank 1 can be used by an application.

## **Special function registers (SFR)**

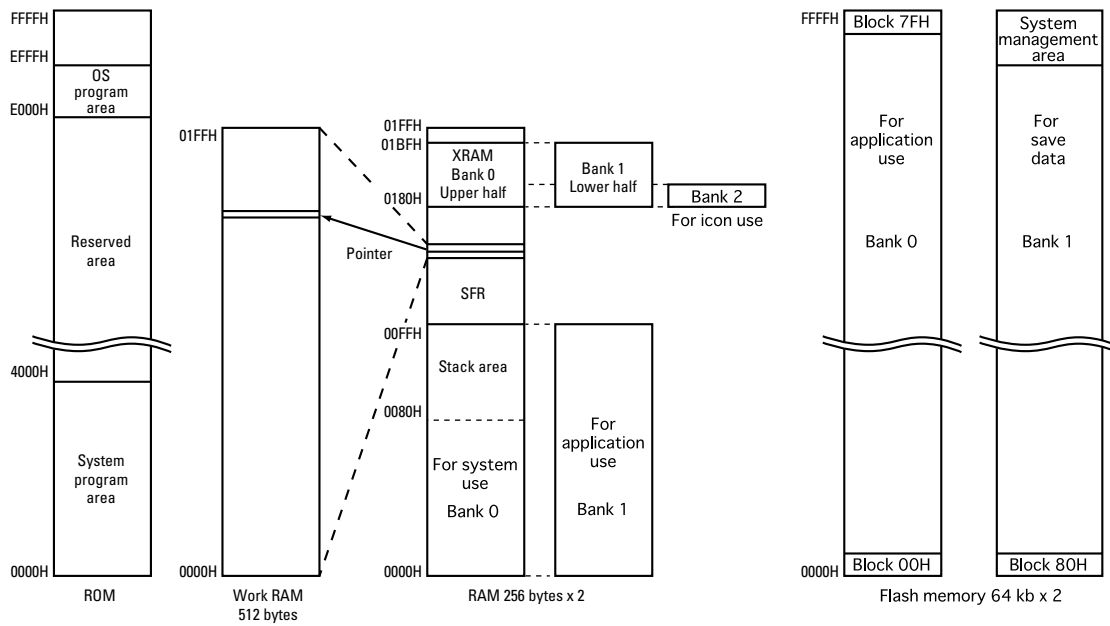
The 100H to 17FH range at the top of RAM is designated as special function registers (SFR). This includes the CPU registers, peripheral device control registers, and other registers.

## **Work RAM (VTRBF)**

When connected to the Dreamcast, the system BIOS uses these 512 bytes as communication buffer. During standalone operation, the memory is available to the application as RAM. Access is possible only through SFR in 1-byte units.

## **XRAM**

This is the RAM for the liquid-crystal display. It corresponds to the video RAM in a conventional computer. XRAM consists of three banks. Banks 0 and 1 can be used by the application to drive the dot-matrix display. Bank 2 serves for the VMU mode icons and cannot be accessed by the application.



**Figure 2.89** VMU Memory Map



# ***System BIOS Functions***

---

Applications can call subroutines that are part of OS programs making up the system BIOS. The system BIOS has the following functions.

## **System initialization**

This is performed when the VMU is reset.

## **Execution mode selection**

This includes game data and application management, editing, application startup and shutdown, time display and adjustment.

Mode selection is performed with the MODE button and the A button.

## **Subroutines**

Subroutines can be used by applications. The subroutines allow flash memory access and readout of internal clock data.

- 1) Flash memory write
- 2) Flash memory read
- 3) Flash memory verify
- 4) Clock count-up timer



# Subroutine Call Procedure

The illustration below shows the operation flow for an application calling a subroutine (part of an OS program), until the return to the application.

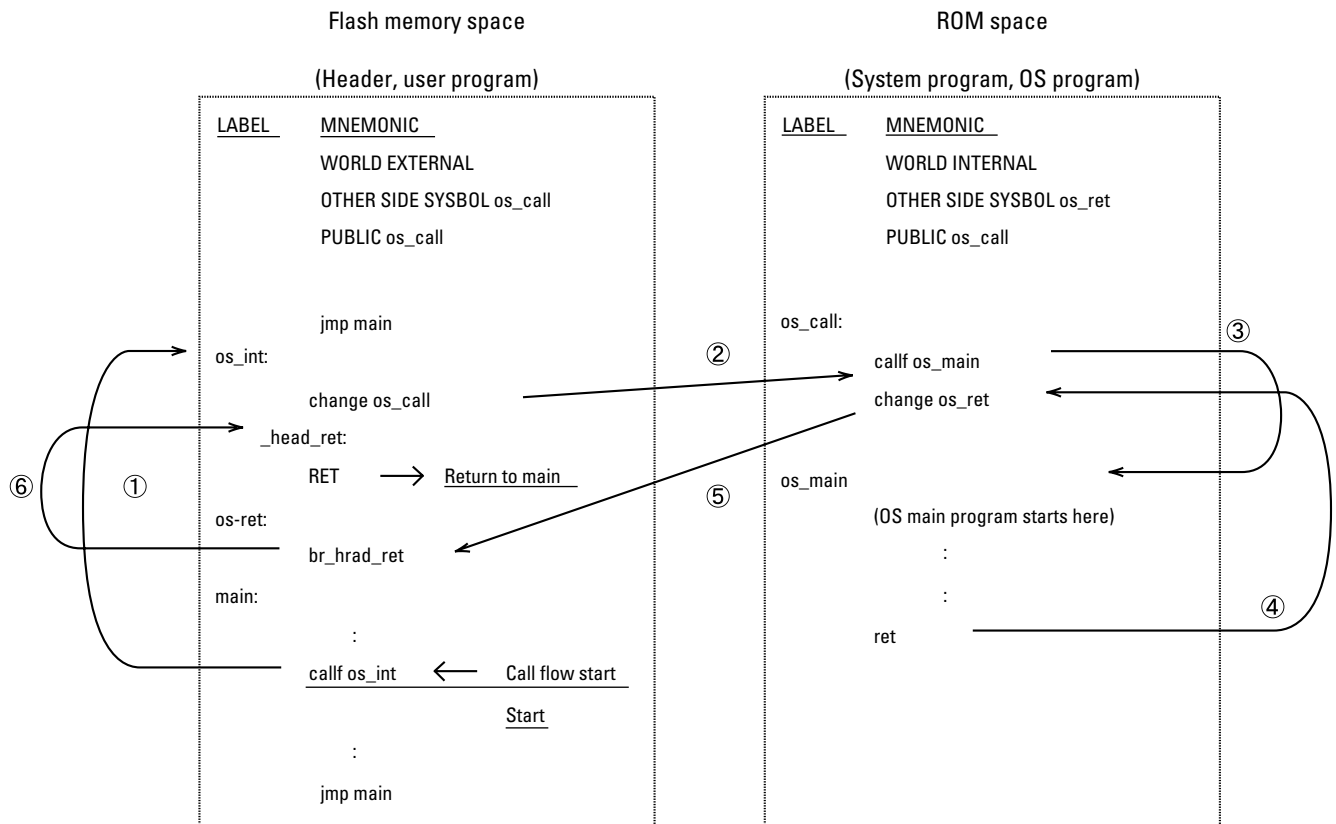


Figure 2.90 Program Call Flow

## Processing Contents of Labels

The purpose and function of labels in subroutines are explained below.

---

**Reference:** When reading this section, you should also refer to "GHEAD.ASM" supplied with the VMU SDK.

---

---

**Caution:** In the explanation below, labels are referred to using preliminary names.

---

### **Application (flash memory)**

#### **main**

Application main program

#### **os\_int**

Subroutine for switching to ROM space processing.

In the example, calling this subroutine will cause the program to move to ROM space processing. When returning from ROM space, the main program resumes.

This subroutine is provided in the header.

#### **os\_ret**

Subroutine for returning to the flash memory space.

While processing occurs in ROM space, executing a CHANGE instruction with this label as argument causes a return to the flash memory. After the return, processing jumps to the interrupt return routine provided in the header.

### **System BIOS (ROM)**

#### **os\_CALL**

This is a return routine for calling an OS program and returning to flash memory. It calls a subroutine in an OS program and returns processing to the flash memory space after completion of the subroutine.

#### **os\_main**

This is the main OS program. It performs processing for the provided subroutines.

## Interaction Between System BIOS and Application

Assuming that an application in the flash memory is running, the processing flow from calling an OS program until return is described below. Refer also to the sample flow chart.

- 1) At the point where the running application wants to use an OS program, it calls the `os_int` subroutine.
- 2) Interrupt processing routines which need to jump to an OS program must contain the `os_int` subroutine.
- 3) The `CHANGE` instruction in the `os_int` subroutine jumps to the OS program call routine in ROM (`os_CALL`).
- 4) The OS program call routine calls the subroutine in the OS main program (`os_main`). OS program processing begins at this point.
- 5) When OS program processing ends, the `RET` instruction jumps to the next address of the `CALL` instruction in the OS program call routine. The OS program call routine always must contain a `CHANGE` instruction for returning to the flash memory after the OS program `CALL` instruction.
- 6) After returning from the OS program subroutine, the `CHANGE` instruction moves processing to the flash memory. The application provides a subroutine (`os_ret`) to be called when returning from ROM.
- 7) This subroutine is called a header. It is supplied as part of the library provided to developers, and must be placed at a fixed location in the application.
- 8) (In the current example, the headers `os_int` and `os_ret` are used.)
- 9) From the above return routine, processing returns to the `os_int` subroutine and then to the main program (`main`) through the `RET` instruction.

---

**Note: CHANGE instruction**

The `CHANGE` instruction is used to move from the flash memory space to the ROM space and vice versa. Executing the `CHANGE` instruction causes processing of a program currently running in ROM (or flash memory) to change to flash memory (or ROM). The program counter is set to the specified label (or address).

---



# Application Shutdown Procedure When MODE Button is Pressed

If the MODE button is pressed while an application is running, processing must be interrupted and the system application mode management screen must be restored immediately.

This section explains the procedure for handing over control from the game application to the system application.

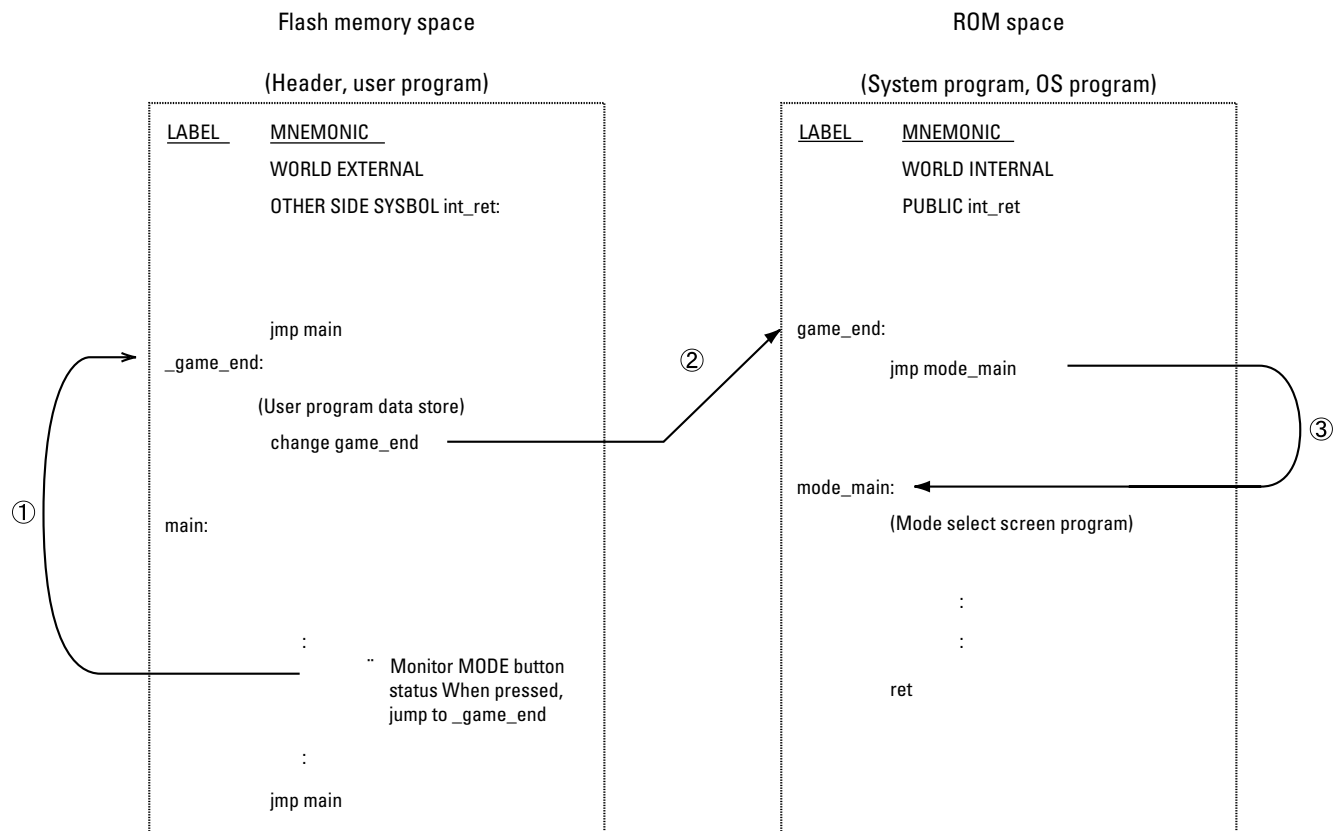


Figure 2.91 Mode Select Screen Restore Flow

# Processing Contents of Labels

The purpose and function of labels in subroutines are explained below.

---

**Reference:** When reading this section, you should also refer to “GHEAD.ASM” supplied with the VMU SDK.

---

---

**Caution:** In the explanation below, labels are referred to using preliminary names, except for `game_end`.

---

### **Application (flash memory)**

#### **main**

Application main program

The application must be programmed to jump to the OS program return routine listed below when the MODE button is pressed.

#### **`_game_end`**

Subroutine for terminating the application and moving processing to the OS program. If data for the application are to be saved, the code for saving data must be included before jumping to this subroutine.

---

**Caution:** The OS program does not save data.

---

### **System BIOS (ROM)**

#### **`game_end`**

This subroutine serves as a window for returning to the system BIOS after the application ends. The subroutine starts the mode selection program.

---

**Caution:** All applications must be designed to speedily CHANGE to `game_end` when the MODE button is pressed.  
Any data required for returning to the game at a later point must be saved by the application in flash memory. This must be performed before the CHANGE to `game_end`. The restore procedure for saved data must also be handled by the application.

---

#### **`mode_main`**

This is the mode selection program.

---

**Reference:** For details on mode selection, refer to chapter 19 “VMU Mode Selection” in the appendix.

---



## **Interaction Between System BIOS and Application**

Assuming that an application in the flash memory is running, the processing flow for returning to the mode selection screen is described below. Refer also to the sample flow chart.

1. At the point where the MODE button is pressed while the application is running, processing jumps to the `_game_end` subroutine.

The CHANGE instruction in the `_game_end` subroutine hands processing over to the program in ROM. If data for the application are to be saved, the code for saving data must be included before executing the CHANGE instruction in the `_game_end` subroutine.

---

**Caution:** Do not use the port 3 interrupt for detecting a MODE button press. If the port 3 interrupt processing routine contains a `_game_end` subroutine, the BIOS does not operate normally.

---

2. When processing jumps from the application to the `_game_end` subroutine, the CHANGE instruction in the `_game_end` subroutine moves processing to the `game_end` subroutine in the ROM program.

3. After processing has changed from the flash memory to the `game_end` subroutine, the mode selection program is started.



# ***VMU Initialization***

---

The VMU is automatically initialized in the following cases.

- 1) Unit is connected to Dreamcast, and Dreamcast is turned ON.
- 2) Reset switch on VMU is pressed.
- 3) Batteries are inserted.

The initialization routine includes the following steps.

## **Clear RAM**

The entire contents of RAM (banks 0 and 1) are set to 00H. The contents of XRAM are not changed.

A hardware reset is applied to all registers, and then software initialization is carried out. For information on initial register values after a hardware reset, refer to section “Reset” in the “Hardware” part of this manual.

## **Set system clock and cycle time**

The system clock is set to the quartz oscillator. The cycle time is set to 1/6 of the system clock.

## **Set base timer**

The 14-bit base timer mode is selected, and the base timer clock is set to the quartz oscillator.

The base timer interrupt is enabled and counting starts.

## **Set master interrupt**

The master interrupt is enabled.

**Set LCD driver**

The LCD controller is activated, and the LCD clock is set to 1/2 of the clock signal input to the LCD driver. The display start address is set to 00H in XRAM, and the character count register and time division register are set.

Then the LCD is set to ON.

**Set port 1**

All bits of port 1 are set to input. Bit 7 of port 1 is set to audio output.

---

**Caution:** After initialization, bit 7 of port is in input mode. Therefore it must be again set to output mode by the application.

---

Bits 5 to 0 of port 1 (VMU serial interface) are set to function as synchronous serial interface.

**Set port 3**

All bits of port 3 are pulled up and set to input mode. Port 3 interrupt source generation is enabled, and interrupt request is enabled.

**Initialize Maple bus interface circuit**

The Maple bus interface circuit is initialized.

**Set work RAM**

The work RAM is set to be available to applications.

# ***Subroutine Reference***

---

This section explains the subroutines contained in the system BIOS.

## **Flash Memory Access Functions**

The following subroutines are provided for flash memory access.

### **Flash memory page data read**

Reads 128 bytes of data from the flash memory space.

### **Flash memory write**

Writes 128 bytes of data to the flash memory space.

### **Flash memory verify**

Verifies data written to the flash memory space.

---

**Caution:** When performing flash memory access, the application must switch the system clock to the RC oscillator.

---

Do not switch the clock within GHEAD.ASM.

## Subroutine Use Precautions

When accessing the flash memory space, observe the following precautions.

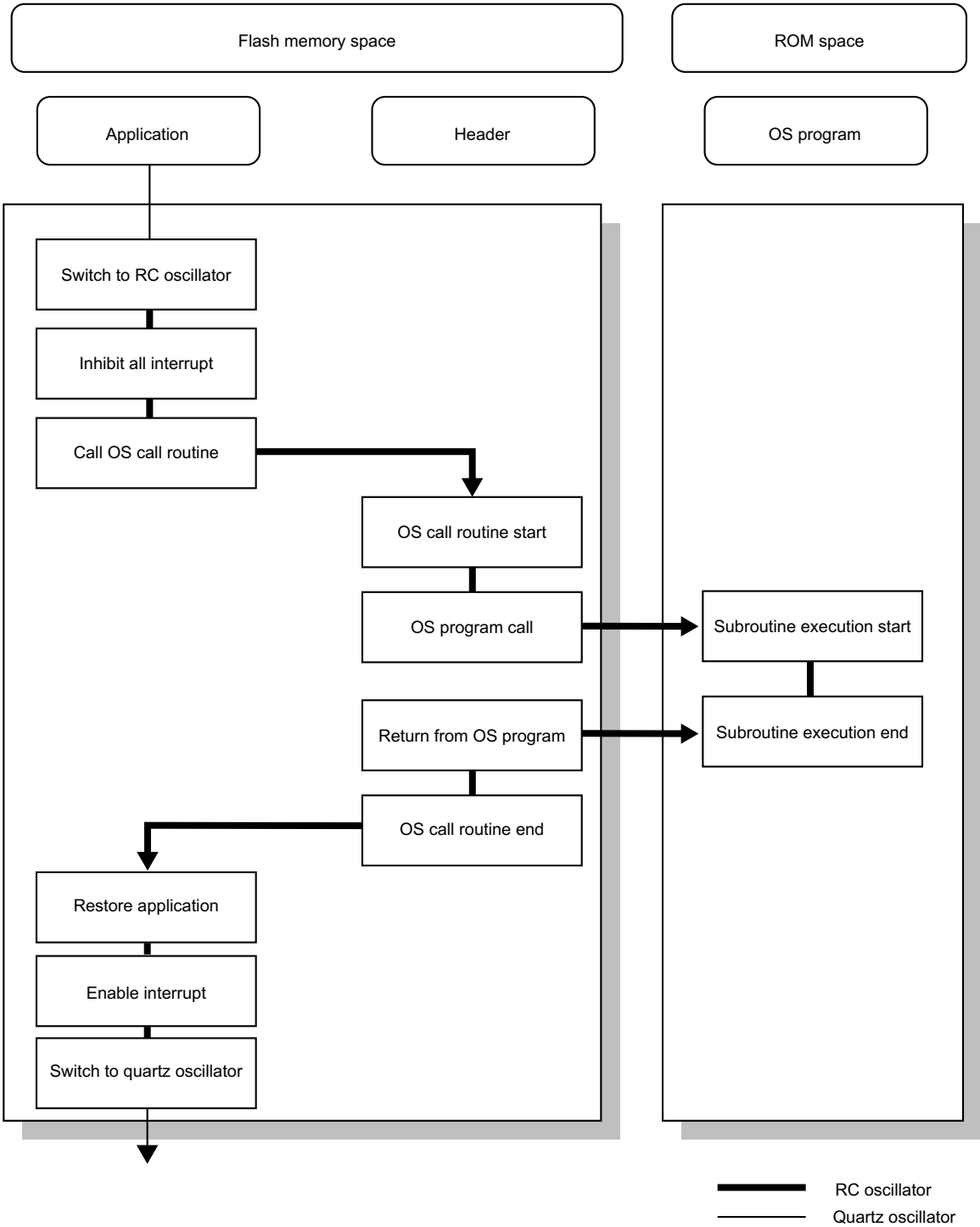
The VMU incorporates three system clock types that can be used to synchronize instruction execution cycles.

In standalone operation, the quartz oscillator is used, but for flash memory write access, the clock must be switched to the RC oscillator with the 1/6 division ratio setting before calling the flash memory access subroutine. For flash read or verify access, the RC oscillator with any division ratio setting can be used.

When switching to the RC oscillator, inhibit all interrupts including the base timer.

After the subroutine is completed, enable all interrupts and switch back to the previously used clock. The proper timing for clock switching is shown below.

System clock oscillator source	Frequency	Instruction cycle time
RC oscillator	879.236 kHz	6.284 ms
Quartz oscillator	32.768 kHz	183.105 ms



**Figure 2.92** Clock Switching Flow for Flash Memory Access

## Flash memory routines

### fm\_prd\_ex(ORG 0120H)

#### Flash memory page data read

##### Arguments

Flash memory read start upper address: fmadd\_h (RAM bank 1 07EH)

Flash memory read start lower address: fmadd\_l (RAM bank 1 07FH)

Flash memory read bank address: fmbank (RAM bank 1 07DH)

##### Return values

Read data (128 bytes): RAM bank 1 080H to 0FFH

##### Broken registers

When this subroutine is called, the following registers are broken.

ACC, TRL, TRH, r0

##### Function

Reads one continuous page of data (128 bytes) starting at the specified address in flash memory.

##### Description

By calling this subroutine, one continuous page of data (128 bytes) can be read from the flash memory. For using the subroutine, the following settings must be made beforehand.

---

**Caution:** This subroutine does not return error information. Make sure that arguments are specified correctly. Call this subroutine only when STAD is set to 00H. If called while STAD is set to a value other than 00H, a part of the screen will be rewritten.

---

##### RAM bank settings

1) Set RAM bank to "1" (set bit 1 of PSW to "1")

---

**Note:** For information on the PSW register, refer to section 3.8 "Program Status Word (PSW)" in the "Hardware" part of this manual.

---

##### Set flash memory read start address

2) Set upper address (8 bits): fmadd\_h (RAM bank 1 07EH)

3) Set lower address (8 bits): fmadd\_l (RAM bank 1 07FH)



**Set flash memory read bank**

- 4) Set read flash memory bank to bank 0.  
Set RAM bank 1 07DH to 00H.

**Caution:** If another value than the above is set, normal operation is not assured.

The read data are written to RAM bank 1 080H to 0FFH.

**Caution:** When making the read settings, observe the following points.

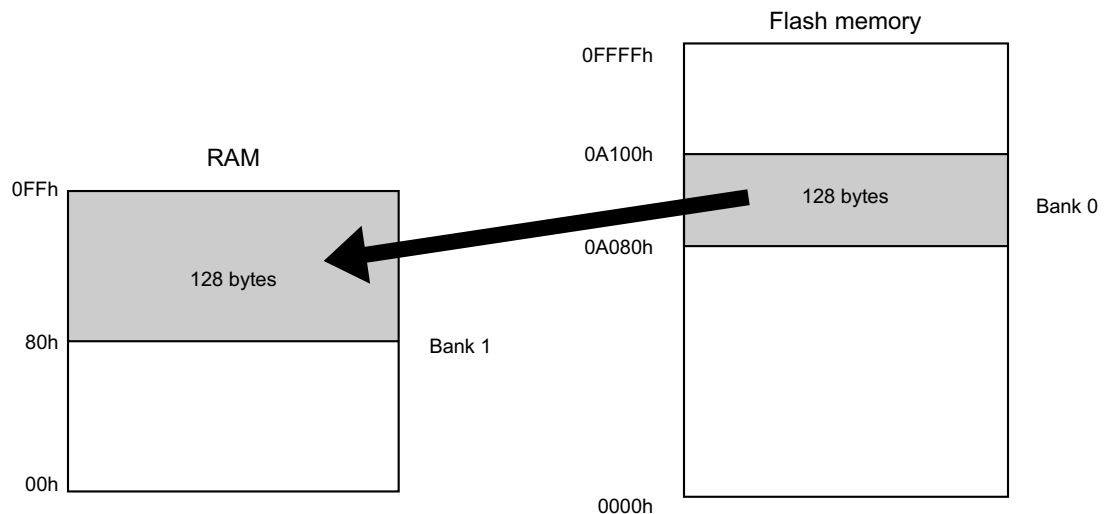
- Data spanning two pages cannot be read. The read start address must be set to the start of a page. The start address of a page can be determined as follows.  
Start address value (2 bytes) = 080H x page number (0 to 511)  
Because data are read in units of one page, bits 0 - 6 of the lower address must be set to "0". If set to an address different from the start address, normal operation is not assured.
- The read data overwrite the original location in RAM.

**Note: About pages**

The flash memory space is divided into pages of 128 bytes each. The flash memory is managed using these page units. Because the size of one bank in the flash memory is 64 KB, a bank contains 512 pages.

The operation when `fm_prd_ex` executes is shown below.

\* When set to  
`fmadd_h = A0h`  
`fmadd_l = 80h` (page no. 321)



**Figure 2.93** Data Transfer With `fm_prd_ex`

### fm\_wrt\_ex(ORG 0100H)

## Flash memory data write

### Arguments

Flash memory write start upper address: fmadd\_h (RAM bank 1 07EH)

Flash memory write start lower address: fmadd\_l (RAM bank 1 07FH)

Flash memory write bank address: fmbank (RAM bank 1 07DH)

Flash memory write data (128 bytes): RAM bank 1 080H to 0FFH

### Return values

ACC At normal end, 00H is set in the accumulator. At abnormal end, 0FFH is set in the accumulator.

### Broken registers

When this subroutine is called, the following registers are broken.

ACC, B, C, TRL, TRH, r0

### Function

Writes one continuous page of data (128 bytes) starting at the specified address in flash memory.

### Description

By calling this subroutine, one continuous page of data (128 bytes) can be written to the flash memory. For using the subroutine, the following settings must be made beforehand.

- 1) Set bit 1 of PSW to "1", to select RAM bank 1.
- 2) Store data to write to flash memory in RAM bank 1 080H - 0FFH.
- 3) Set 07DH of RAM bank 1 to 00H, to set write flash memory to bank 0.

---

**Caution:** Flash memory bank 1 may not be accessed by applications. Do not write any data to this bank.

---

- 4) Write flash memory upper address (8 bits) to 07EH of RAM bank 1, and write lower address (8 bits) to 07FH of RAM bank 1.

---

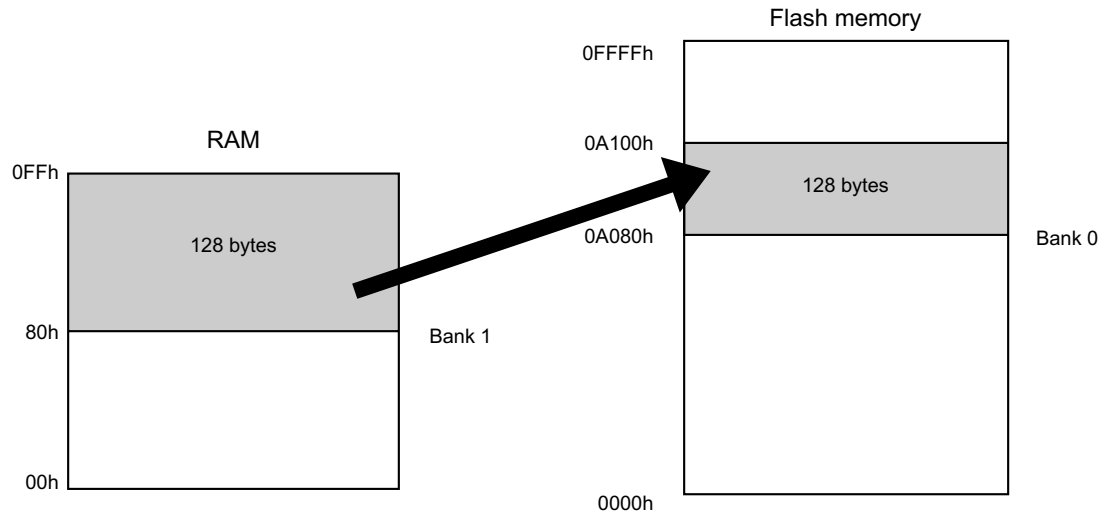
**Caution:** When making the write settings, observe the following points.

- Data spanning two pages cannot be written. The write start address must be set to the start of a page. The start address of a page can be determined as follows.  
Start address value (2 bytes) = 080H x page number (0 to 511)  
Because data are written in units of one page, bits 0 - 6 of the lower address must be set to "0". If set to an address different from the start address, normal operation is not assured.
- Call only when STAD is set to 00H. If called while STAD is set to a value other than 00H, a part of the screen will be rewritten.
- Switch the system clock to RC oscillator with the 1/6 division ratio setting.

---

The operation when `fm_wrt_ex` executes is shown below.

\* When set to  
`fmadd_h = A0h`  
`fmadd_l = 80h` (page no. 321)



**Figure 2.94** Data Transfer With `fm_wrt_ex`

## **fm\_vrf\_ex(ORG 0110H)**

### **Flash memory page data verify**

#### **Arguments**

Flash memory verify start upper address: `fmadd_h` (RAM bank 1 07EH)

Flash memory verify start lower address: `fmadd_l` (RAM bank 1 07FH)

Flash memory verify bank address: `fmбанк` (RAM bank 1 07DH)

Flash memory verify data (128 bytes): RAM bank 1 080H - 0FFH

#### **Return values**

The verify result is set in the accumulator. If there was no mismatch, 00H is set. If there was a mismatch, a value other than 00H is set.

#### **Broken registers**

When this subroutine is called, the following registers are broken.

ACC, TRL, TRH, r0

#### **Function**

After writing data to flash memory, this function checks whether the data were written correctly. Use the function after using `fm_wrt_ex` to write tot flash memory.

### **Description**

This subroutine compares the 128 bytes of data specified when calling `fm_wrt_ex` to the data actually written to the flash memory.

---

**Caution:** Call this subroutine only when STAD is set to 00H. If called while STAD is set to a value other than 00H, a part of the screen will be rewritten.

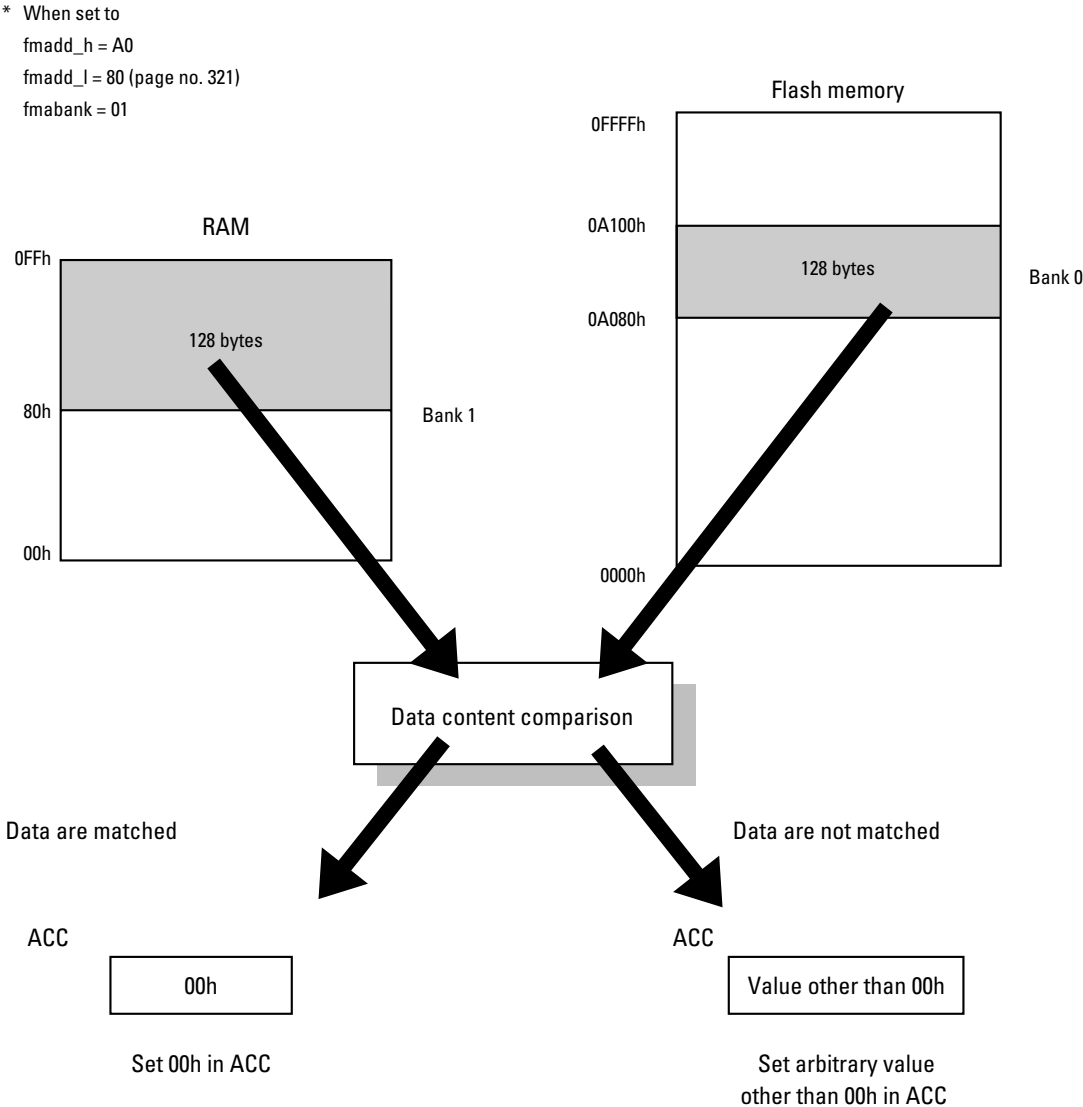
---

This subroutine may therefore only be called immediately after calling the `fm_wrt_ex` subroutine.

When calling the subroutine, the same arguments as for the `fm_wrt_ex` subroutine must be supplied. If different arguments are supplied, data verify will not yield correct results.

After calling the subroutine, 00H is set in the accumulator if all 128 bytes of data were matched. If there was a mismatch, a value other than 00H is set.

The operation when `fm_vrf_ex` executes is shown below.



**Figure 2.95** Execution of *fm\_vrf\_ex*

## Clock Function

### timer\_ex

#### Clock count-up timer

##### Arguments

None

##### Return values

Year :	year_h	(RAM bank 0 017HÅC18H)
Month :	mon_h	(RAM bank 0 019H)
Day :	day_h	(RAM bank 0 01AH)
Hour :	hour_h	(RAM bank 0 01BH)
Minute :	min_h	(RAM bank 0 01CH)
Second :	sec_h	(RAM bank 0 01DH)

The year data use 2 bytes. The upper byte is stored in 17H and the lower byte in 18H. RAM bank 0 017H is assigned to year\_h. When accessing address 018H, the address for year\_h + 1 must be accessed.

---

**Caution:** The time data obtained by this subroutine are all in hexadecimal format. They must be converted to decimal format by the application.

---

The work area comprises a BCD date area, but this area is not updated by timer\_ex.

##### Function

Gets current date and time data and places them in the specified area of RAM bank 0.

##### Description

This subroutine is a timer/counter using the base timer interrupt.

---

**Caution:** The base timer interrupt uses timer\_ex. For using the base timer interrupt from the application, call the user-side handler immediately after the label timer\_ex\_exit in GHEAD.ASM. Call this subroutine after generating a base timer interrupt source and jumping to the interrupt vector. At this time, be sure to reset the base timer interrupt source (BTCR1 = 0).

---

If the interrupt source is not reset, the clock function will not work properly.

# ***Low Battery Voltage Auto Detection***

---

An automatic low battery voltage detection function which displays a warning message on the LCD is incorporated in the system BIOS.

Actions which cause high power consumption such as flash memory data write or data transfer to another VMU may falsely trigger the warning. Therefore the detection function should be disabled before carrying out such actions.

## **Low battery voltage auto detection flag**

This flag specifies whether low battery voltage auto detection is performed or not. Applications can manipulate this flag.

---

**Caution:** Be sure to set the low battery voltage auto detection flag to 0FFH (off) before having an application perform one of the following functions. Otherwise the high current consumption caused by these functions may falsely trigger the low battery voltage auto detection.

- Communication with other VMU via serial interface
- Writing to flash memory

---

## **Address**

06EH (RAM bank 0) Low battery voltage auto detection flag

When set to 00H, low battery voltage auto detection is carried out. When set to 0FFH, low battery voltage auto detection is not carried out.

---

**Caution:** Do not set the flag to values other than 00H or 0FFH.

---

### **Operation**

The low battery voltage auto detection function monitors the battery voltage. When it falls below a certain threshold, the function interrupts the currently running program and displays a warning message for 3 seconds on the LCD.

### **Description**

The low battery voltage auto detection function comprises code for both voltage detection and message display. When the low battery voltage auto detection flag is set to 00H, these functions are carried out automatically, regardless of the operation status of the VMU. When the low battery voltage auto detection flag is set to 0FFH, all functions related to automatic low battery voltage detection are turned off.

---

**Caution:** Programs to save data in flash memory when low voltage is detected should monitor the low voltage detection flag (bit 1 of port 7) rather than using the low voltage interrupt. If the low voltage interrupt is used, triggering may occur while writing to flash memory or during serial communication, although the battery voltage is still sufficient.

---

---

**Note:** For information on the low voltage detection flag, refer to section on "Port 7" in the "Hardware" part of this manual.

---



# ***List of Defined Variables***

The following variables are required for using the OS program BIOS.

## **Time data variables**

<b>Symbol</b>	<b>Address (RAM bank)</b>	<b>Contents</b>	<b>Comment</b>
year	010H (Bank 0)	Year (BCD 4 digits)	Not updated by timer_ex
mon	012H (Bank 0)	Month (BCD 2 digits)	Not updated by timer_ex
day	013H (Bank 0)	Day (BCD 2 digits)	Not updated by timer_ex
hour	014H (Bank 0)	Hours (BCD 2 digits)	Not updated by timer_ex
min	015H (Bank 0)	Minutes (BCD 2 digits)	Not updated by timer_ex
sec	016H (Bank 0)	Seconds (BCD 2 digits)	Not updated by timer_ex
year_h	017H (Bank 0)	Year (HEX 4 digits)	
mon_h	019H (Bank 0)	Month (HEX 2 digits)	
day_h	01AH (Bank 0)	Day (HEX 2 digits)	
hour_h	01BH (Bank 0)	Hours (HEX 2 digits)	
min_h	01CH (Bank 0)	Minutes (HEX 2 digits)	
sec_h	01DH (Bank 0)	Seconds (HEX 2 digits)	
sec_f	01EH (Bank 0)	Work area	Use prohibited
leaf_f	01FH (Bank 0)	Work area	Use prohibited

**Caution:** The BCD data fields year, mon, day, hour, min, sec are a work area for applications accessing the clock function. Because `timer_ex` does not perform BCD conversion, this work area is not updated.

---

**Low battery voltage detection variables**

Symbol	Address (RAM bank)	Content
None	06EH (Bank 0)	Low battery auto detect flag
00: Auto detect on		
FFH: Auto detect off		

To perform low-battery checking from the application without using auto detect, check bit 1 of port 7.

**Flash memory variables**

Symbol	Address (RAM bank)	Content
Fmbank		
fmadd_h		
fmadd_l	07DH (bank 1)	
07EH (bank 1)		
07FH (bank 1)	Specify flash memory bank	
Flash memory address (upper 8 bits)		
Flash memory address (lower 8 bit)		

---

# Sound Output Method

---

This section describes the VMU sound output method. The sound output uses timer 1.

## Timer 1 Outline

This section describes the timer 1 used for VMU sound output. The timer 1 incorporated in the VMU is a 16-bit timer with the following four functions.

Mode 0: 8 bit reload timer x 2 channels

Mode 1: 8 bit reload timer + 8 bit pulse generator

Mode 2: 16 bit reload timer

Mode 3: Variable bit length pulse generator (9 to 16 bits)

VMU uses mode 1 for producing sound. For information on use of the other modes, refer to section on “Timer 1 (T1)” in the “Hardware” part of this manual.

## Timer 1 Block Configuration

The timer 1 used for VMU sound output has the following block configuration.

### Timer 1 low (T1L)... ②

This is an 8 bit reload timer which uses the cycle clock or 1/2 the cycle clock as clock.

At T1L overflow, the T1LR data are reloaded, and sent to T1L if T1LRUN (T1CNT bit 6) is set to “0”.

### Timer 1 low comparator (T1LC)... †

This comparator consists of the 8 bit timer 1 low comparison data register (T1LC) and an 8 bit data comparison circuit. It serves to compare the T1L and T1LC data.

### Timer 1 high (T1H)... ➡

This is an 8 bit reload timer which uses the cycle clock or the T1L overflow as clock.

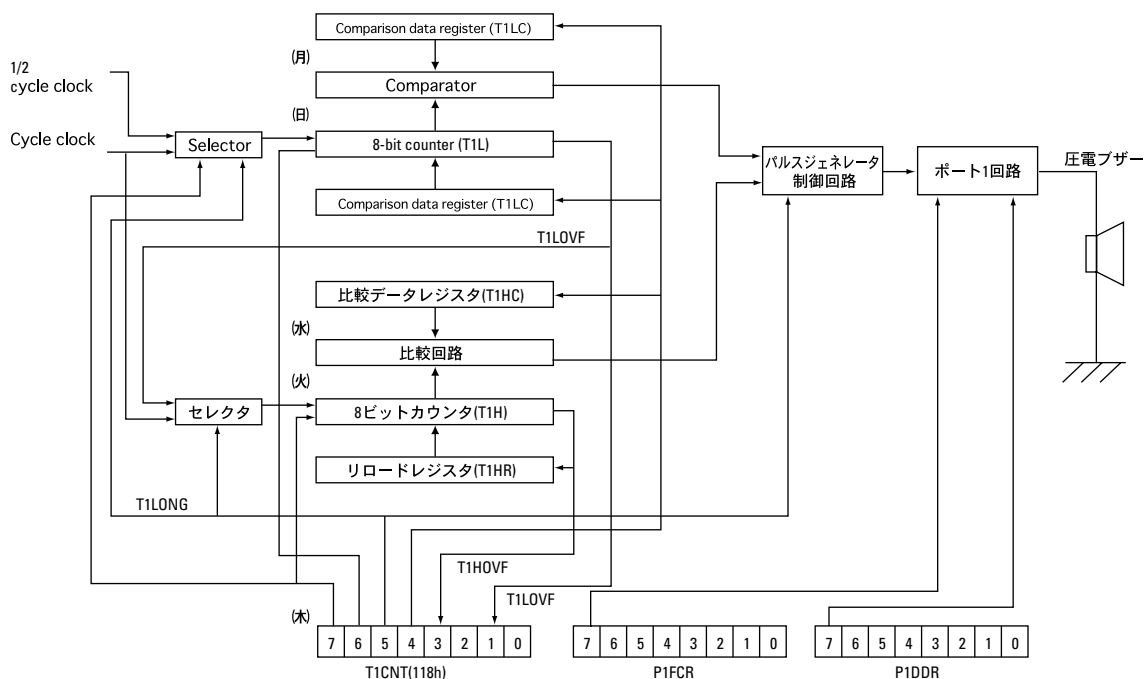
At T1H overflow, T1HR data are reloaded, regardless of whether T1HRUN (T1CNT bit 7) is reset.

**Timer 1 high comparator (T1HC)... ㊦**

This comparator consists of the 8 bit timer 1 high comparison data register (T1HC) and an 8 bit data comparison circuit. It serves to compare the T1H and T1HC data.

**Timer 1 control register (T1CNT)... ㊦**

Controls timer 1 mode setting and interrupt control.



**Figure 2.96** Timer 1 Block Diagram

**Related Registers**

To control timer 1, the following registers must be controlled.

Symbol	Address	Function
T1L	11BH	Timer 1 lower counter register
T1LR	11BH	Timer 1 lower reload register
T1LC	11AH	Timer 1 lower comparison data register
T1CNT	118H	Timer 1 control register
P1	114H	Port 1 latch register
P1DDR	145H	Port 1 data direction register
P1FCR	146H	Port 1 control register
OCR	10EH	Oscillation control register

For details on the above registers, refer to the “Hardware” part of this manual.

## Mode Setting

This section explains how to set timer 1 to the mode required for sound output (mode 1). The following four registers are required for the setting.

T1CNT (bit 5: T1LONG)

P1 (bit7: P17)

P1DDR (bit 7: P17DDR)

P1FCR (bit 7: P17FCR)

The register values for each mode are shown below. The available cycle clock setting for each mode is also shown.

Mode	Clock frequency	T1LONG	P17FCR	P17DDR	P17
1	Tcyc	0	1	1	0

Tcyc in the table indicates the clock cycle.

To use the VMU sound output function, be sure to set the system clock to the quartz oscillator. At other settings, correct sound output may not be obtained.

The cycle clock is as follows.

System clock 32.768 kHz (Tcyc = 183.105 ms)

For information on setting the system clock, refer to the “Hardware” part of this manual.

---

**Caution:** Problems when using other system clock settings  
If the sound output function is used while the system clock is set to the RC oscillator, the tolerances of the RC oscillator will adversely affect the sound output. Be sure to use the quartz oscillator.

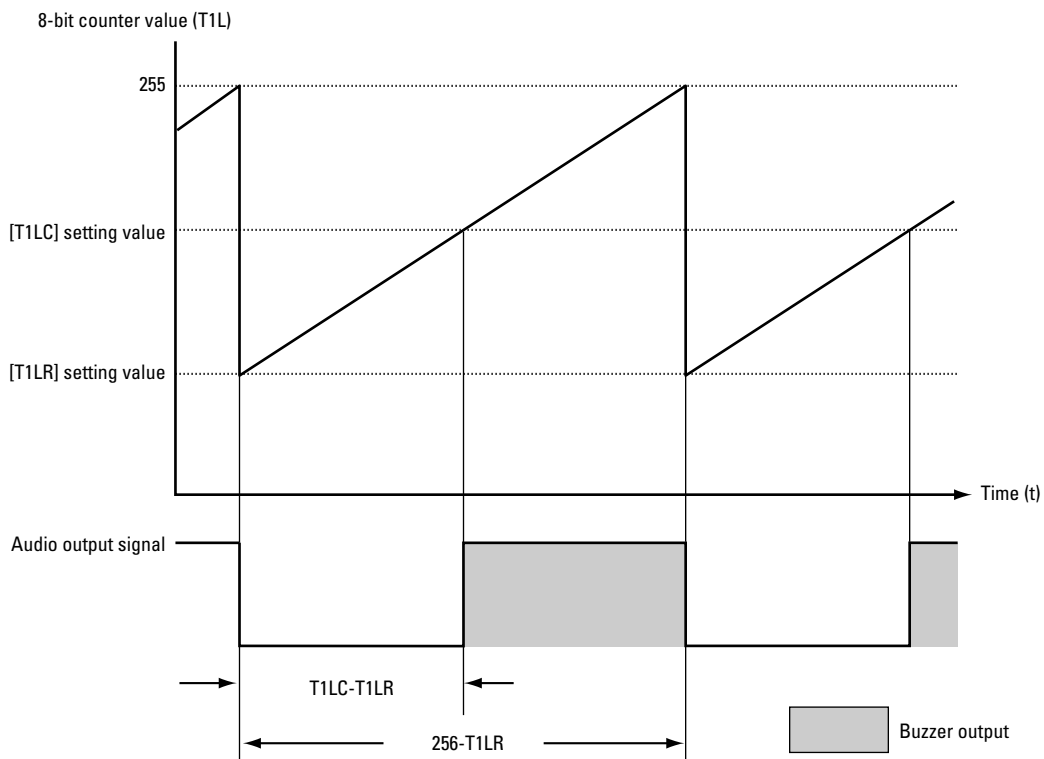
---

## 8 Bit Counter Mode

This section explains VMU sound output using the 8 bit counter mode. For information on basic operation, refer to the "Hardware" part of this manual.

### Output Waveform and Parameter Setting

This section describes the waveform of the signal that can be output in 8 bit counter mode and the available parameters.



**Figure 2.97** *Output Waveform*

## 8 Bit Counter Mode Setting

This section describes sound output in 8 bit counter mode. To use sound output in 8 bit counter mode, make the following settings.

### 1 Output waveform setting

Set the parameters (T1LR, T1LC) to obtain the desired waveform. Use equations (1) and (2) shown below to determine the waveform.

Audio output signal "Low" level pulse width (decimal) = (T1LC setting value ~ T1LR setting value) x Tcyc... (1)

Output signal frequency (decimal) = (256 - T1LR setting value) x Tcyc... (2)

Tcyc: cycle clock

For details in output waveform parameter settings, refer to section 15.2.1 "Output Waveform and Parameter Setting".

### Timer 1 mode setting

Set timer 1 to mode 1. The following four registers are required for mode setting.

T1CNT (bit 5: T1LONG)

P1 (bit 7: P17)

P1DDR (bit 7: P17DDR)

P1FCR (bit 7: P17FCR)

The register values for mode 1 are shown below.

Mode	T1LONG	P17FCR	P17DDR	P17
1	0	1	1	0

### Sound start

Timer 1 (lower 8 bit) starts to count, and sound is output. To control timer 1 count start/stop, make the following setting.

#### 1) Waveform parameter update

Set T1CNT bit 4 (ELDT1C) to "1". If this setting is not made, the waveform parameter set with T1LR, T1LC does not become effective.

If the waveform parameter is changed while T1CNT bit4 is "1", the new parameter setting becomes effective immediately.

#### 2) Timer 1 count start

Set T1CNT bit 6 (T1LRUN) to "1".

### Sound stop

To stop sound output in 8 bit counter mode, make the following settings.

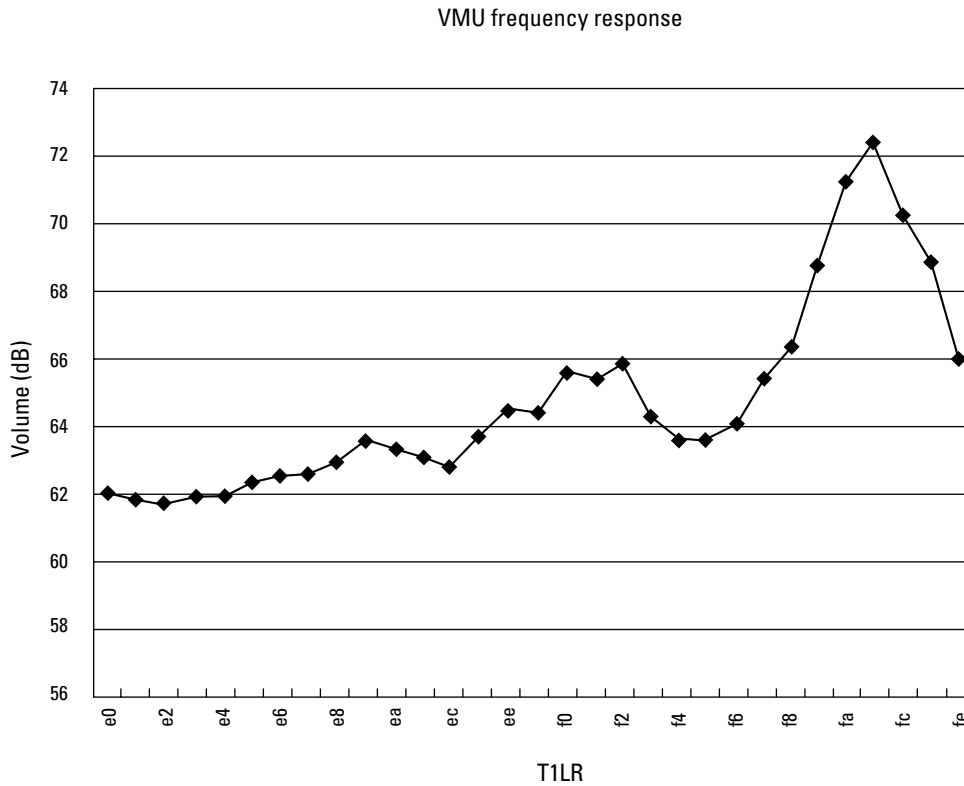
Set timer 1 (T1L) count stop flag (T1CNT bit 6) to "0".

The waveform parameter can be changed also during sound output (while timer 1 operates). To continuously output a different frequency, change the output waveform parameter without stopping timer 1. T1CNT bit 4 (ELDT1C) should always be "1" in this case.

## Frequency Characteristics

The frequency characteristics of VMU sound output are shown below.

The T1LR value indicates the setting value for the available frequency range. For details on the correlation between the value of T1LR and the output frequency, refer to section 15.2.4 "Output Frequency Table"



**Figure 2.98** Frequency Response Characteristics

## Output Frequency Table

The following table shows the frequencies (theoretical values) available with the 32.768 kHz system clock.

Due to buzzer characteristics, not all frequencies can actually be output. Use the recommended frequencies indicated in the table.

The sound output signal "Low" level pulse width is set to 1/2 (duty cycle 50%) of the output signal cycle.



T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)
00	80	21.346	40	94	28.461	80	A8	42.691	C0	E0	85.383
01	80	21.429	41	A0	28.610	81	C0	43.027	C1	E0	86.738
02	81	21.514	42	A1	28.760	82	C1	43.369	C2	E1	88.137
03	81	21.599	43	A1	28.913	83	C1	43.716	C3	E1	89.582
04	82	21.684	44	A2	29.066	84	C2	44.068	C4	E2	91.075
05	82	21.771	45	A2	29.222	85	C2	44.427	C5	E2	92.618
06	83	21.858	46	A3	29.379	86	C3	44.791	C6	E3	94.215
07	83	21.946	47	A3	29.538	87	C3	45.161	C7	E3	95.868
08	84	22.034	48	A4	29.698	88	C4	45.537	C8	E4	97.580
09	84	22.123	49	A4	29.861	89	C4	45.920	C9	E4	99.354
0A	85	22.213	4A	A5	30.025	8A	C5	46.309	CA	E5	101.194
0B	85	22.304	4B	A5	30.191	8B	C5	46.705	CB	E5	103.103
0C	86	22.395	4C	A6	30.358	8C	C6	47.108	CC	E6	105.086
0D	86	22.488	4D	A6	30.528	8D	C6	47.517	CD	E6	107.147
0E	87	22.580	4E	A7	30.699	8E	C7	47.934	CE	E7	109.290
0F	87	22.674	4F	A7	30.873	8F	C7	48.358	CF	E7	111.520
10	88	22.769	50	A8	31.048	90	C8	48.790	D0	E8	113.843
11	88	22.864	51	A8	31.226	91	C8	49.230	D1	E8	116.266
12	89	22.960	52	A9	31.405	92	C9	49.677	D2	E9	118.793
13	89	23.057	53	A9	31.587	93	C9	50.133	D3	E9	121.433
14	8A	23.155	54	AA	31.770	94	CA	50.597	D4	EA	124.193
15	8A	23.253	55	AA	31.956	95	CA	51.070	D5	EA	127.081
16	8B	23.352	56	AB	32.144	96	CB	51.552	D6	EB	130.107
17	8B	23.453	57	AB	32.334	97	CB	52.043	D7	EB	133.280
18	8C	23.554	58	AC	32.527	98	CC	52.543	D8	EC	136.612
19	8C	23.656	59	AC	32.721	99	CC	53.053	D9	EC	140.115
1A	8D	23.759	5A	AD	32.919	9A	CD	53.573	DA	ED	143.802
1B	8D	23.862	5B	AD	33.118	9B	CD	54.104	DB	ED	147.689
1C	8E	23.967	5C	AE	33.320	9C	CE	54.645	DC	EE	151.791
1D	8E	24.073	5D	AE	33.524	9D	CE	55.197	DD	EE	156.128
1E	8F	24.179	5E	AF	33.731	9E	CF	55.760	DE	EF	160.720
1F	8F	24.287	5F	AF	33.941	9F	CF	56.335	DF	EF	165.590
20	90	24.395	60	B0	34.153	A0	D0	56.922	E0	F0	170.765
21	90	24.504	61	B0	34.368	A1	D0	57.521	E1	F0	176.274
22	91	24.615	62	B1	34.585	A2	D1	58.133	E2	F1	182.149
23	91	24.726	63	B1	34.806	A3	D1	58.758	E3	F1	188.430
24	92	24.839	64	B2	35.029	A4	D2	59.397	E4	F2	195.160
25	92	24.952	65	B2	35.255	A5	D2	60.049	E5	F2	202.388
26	93	25.066	66	B3	35.484	A6	D3	60.716	E6	F3	210.172
27	93	25.182	67	B3	35.716	A7	D3	61.399	E7	F3	218.579
28	94	25.299	68	B4	35.951	A8	D4	62.096	E8	F4	227.687
29	94	25.416	69	B4	36.189	A9	D4	62.810	E9	F4	237.586
2A	95	25.535	6A	B5	36.430	AA	D5	63.540	EA	F5	248.385
2B	95	25.655	6B	B5	36.674	AB	D5	64.288	EB	F5	260.213
2C	96	25.776	6C	B6	36.922	AC	D6	65.053	EC	F6	273.224
2D	96	25.898	6D	B6	37.173	AD	D6	65.837	ED	F6	287.604
2E	97	26.021	6E	B7	37.428	AE	D7	66.640	EE	F7	303.582
2F	97	26.146	6F	B7	37.686	AF	D7	67.463	EF	F7	321.440
30	98	26.272	70	B8	37.948	B0	D8	68.306	F0	F8	341.530
31	98	26.398	71	B8	38.213	B1	D8	69.171	F1	F8	364.299
32	99	26.527	72	B9	38.482	B2	D9	70.057	F2	F9	390.320
33	99	26.656	73	B9	38.755	B3	D9	70.967	F3	F9	420.345
34	9A	26.787	74	BA	39.032	B4	DA	71.901	F4	FA	455.373
35	9A	26.919	75	BA	39.313	B5	DA	72.860	F5	FA	496.771
36	9B	27.052	76	BB	39.598	B6	DB	73.844	F6	FB	546.448
37	9B	27.186	77	BB	39.887	B7	DB	74.856	F7	FB	607.165
38	9C	27.322	78	BC	40.180	B8	DC	75.896	F8	FC	683.060
39	9C	27.460	79	BC	40.478	B9	DC	76.965	F9	FC	780.640
3A	9D	27.598	7A	BD	40.780	BA	DD	78.064	FA	FD	910.747
3B	9D	27.738	7B	BD	41.086	BB	DD	79.195	FB	FD	1092.896
3C	9E	27.880	7C	BE	41.398	BC	DE	80.360	FC	FE	1366.120
3D	9E	28.023	7D	BE	41.714	BD	DE	81.559	FD	FE	1821.494
3E	9F	28.167	7E	BF	42.034	BE	DF	82.795	FE	FF	2732.240
3F	9F	28.313	7F	BF	42.360	BF	DF	84.069	FF	FF	5464.481

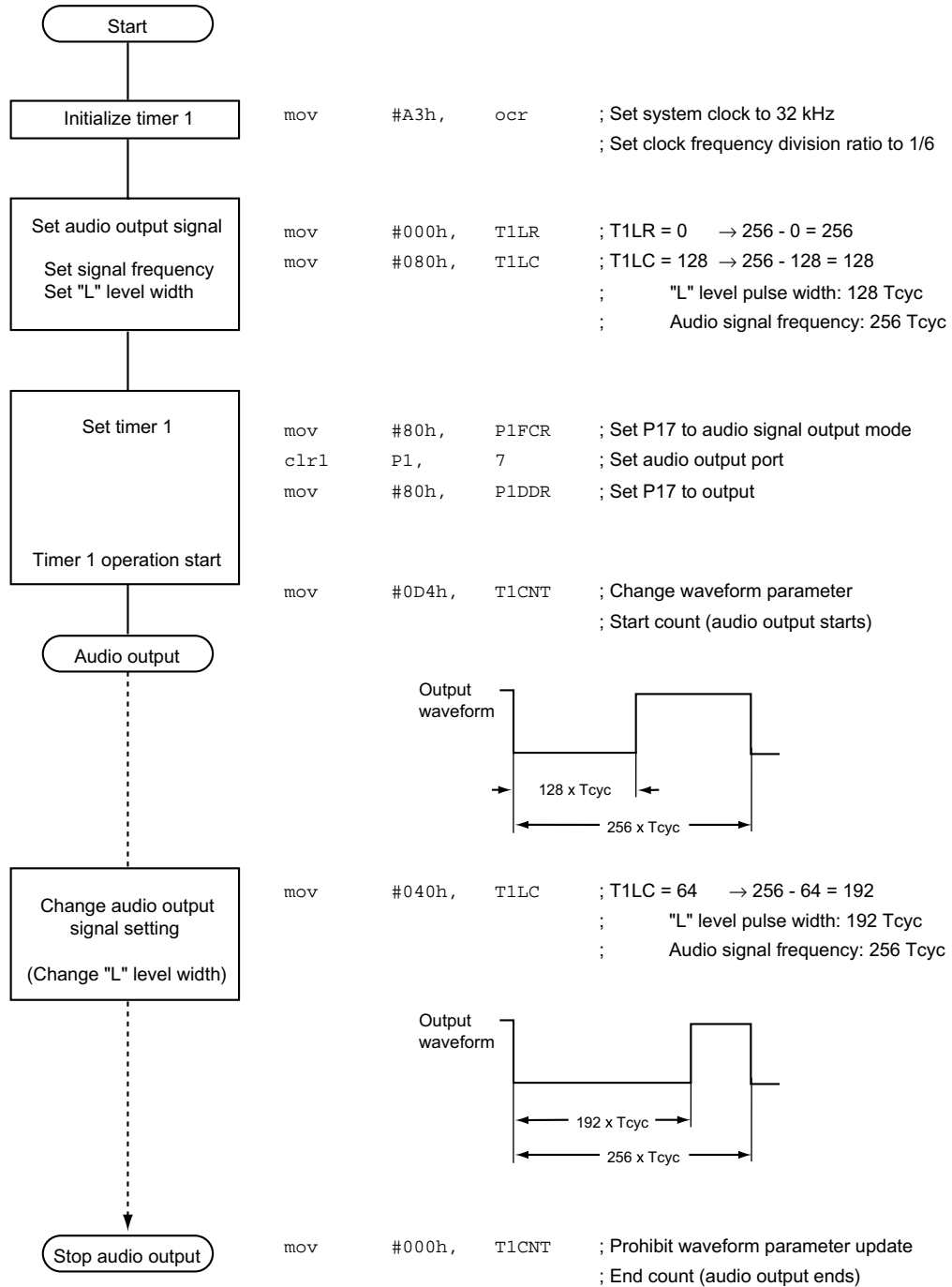
:Recommended settings

**Figure 2.99** Waveform Parameters and Output Frequencies



# ***Sample Program***

---



**Figure 2.100** Flow Chart and Program

# ***Variable Bit Length Pulse Generator***

---

This section provides additional information about the equation shown in “Mode 3: Variable bit length pulse generator (9 to 16 bits)” of section 4.3 “Timer 1 (T1)” in the “Hardware” part of this manual.

- Large interval P cycle  $T_p$

$$T_p = 2[\text{BIT}] \times T_{tc}$$

- Total "L" level pulse width  $\langle \sigma \rangle TL$  of large interval P

$$STL = (2[\text{BIT}] \times T1LC) / 256 + [T1HC] \times T_{tc}$$

- T1HC, T1LC are decimal.
- [T1HC] is the effective number of bits.

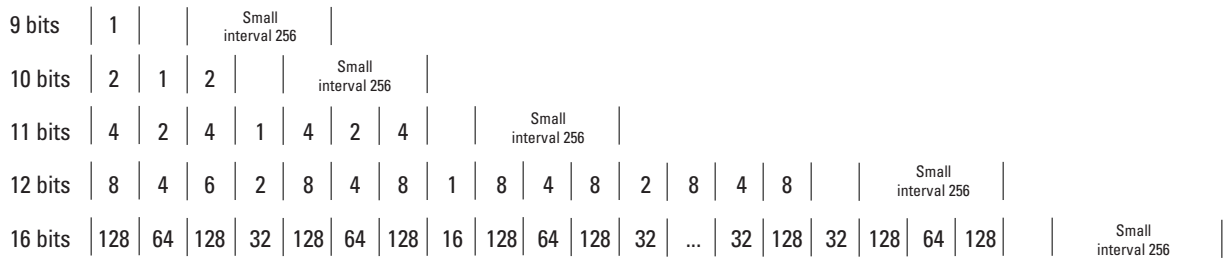
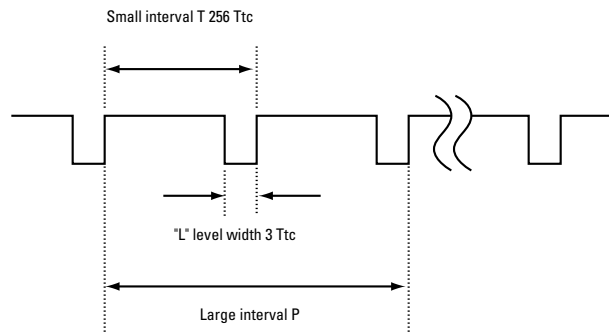
[BIT] is the bit length to be set. The number of small intervals T in the large interval P is determined by the bit length. It is set by the timer 1 high reload register (T1HR) and timer 1 low reload register (T1LR). Set the T1LR to 00H. For a 9-bit pulse generator, the setting is [BIT] = 9, large interval P cycle  $T_p = 29 \times T_{tc} = 512 T_{tc}$ . Since the small interval T is  $256 T_{tc}$  ( $T_{tc}$ : pulse signal clock cycle), T is repeated 2 times in the large interval P.

**Table 2.34**

Bit length	Small interval T repeat count	Pulse generator bit length setting (binary)	"L" level pulse width setting (binary)		
		T1HR value	T1LR value	T1HR value (upper bits)	T1LR value (lower bits)
16	256	0000 0000	0000 0000	XXXX XXXX	XXXX XXXX
15	238	1000 0000	0000 0000	XXXX XXXX	XXXX XXX0
14	64	1100 0000	0000 0000	XXXX XXXX	XXXX XX00
13	32	1110 0000	0000 0000	XXXX XXXX	XXXX X000
12	16	1111 0000	0000 0000	XXXX XXXX	XXXX 0000
11	8	1111 1000	0000 0000	XXXX XXXX	XXX0 0000
10	4	1111 1100	0000 0000	XXXX XXXX	XX00 0000
9	2	1111 1110	0000 0000	XXXX XXXX	X000 0000

(X: 0 or 1) indicates effective bits

[T1LR] specifies the timer 1 low comparison data register (T1LC) value. It sets the "Low" level pulse width common to all small intervals. If [T1LC] = 3, a "Low" level pulse of 3 Ttc is applied to all small intervals, and the "Low" level pulse width of the large interval is 6 Ttc (if [T1HC] = 0).

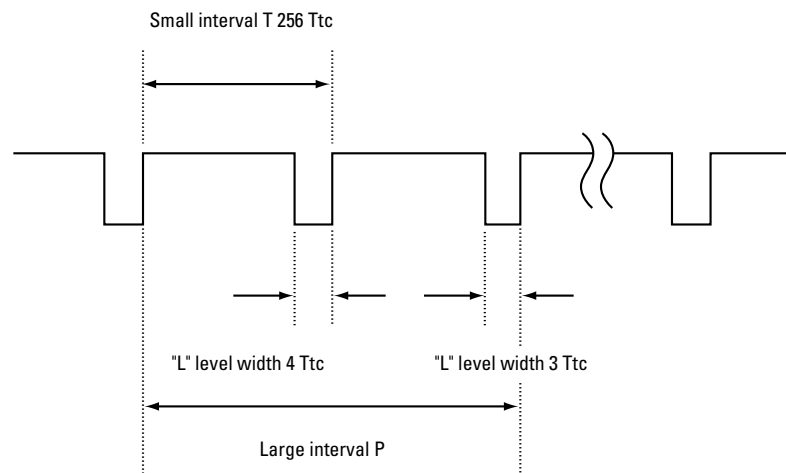


**Figure 2.101** "L" Level Width for T1LC = 3

[T1HC] specifies the effective value of the timer 1 high comparison data register (T1HC), determining the number of "Low" level pulses to be added in the large interval P.

The position of the small intervals to which "Low" level pulses are added by [T1HC] is shown in the illustration below. If [T1HC] = 1, a "Low" level pulse of 1 Ttc is added to the intervals marked "1". If [T1HC] = 6, a "Low" level pulse of 1 Ttc is added to the intervals marked "2" and "4".

In the above example, [T2HC] = 0 applies. If [T2HC] = 1, the situation is as follows, with the "Low" level pulse width in the large interval P being 7 Ttc. For a 9-bit pulse generator, the effective bit is 1.



**Figure 2.102** "L" Level Width (7 Ttc) for T1HC = 1

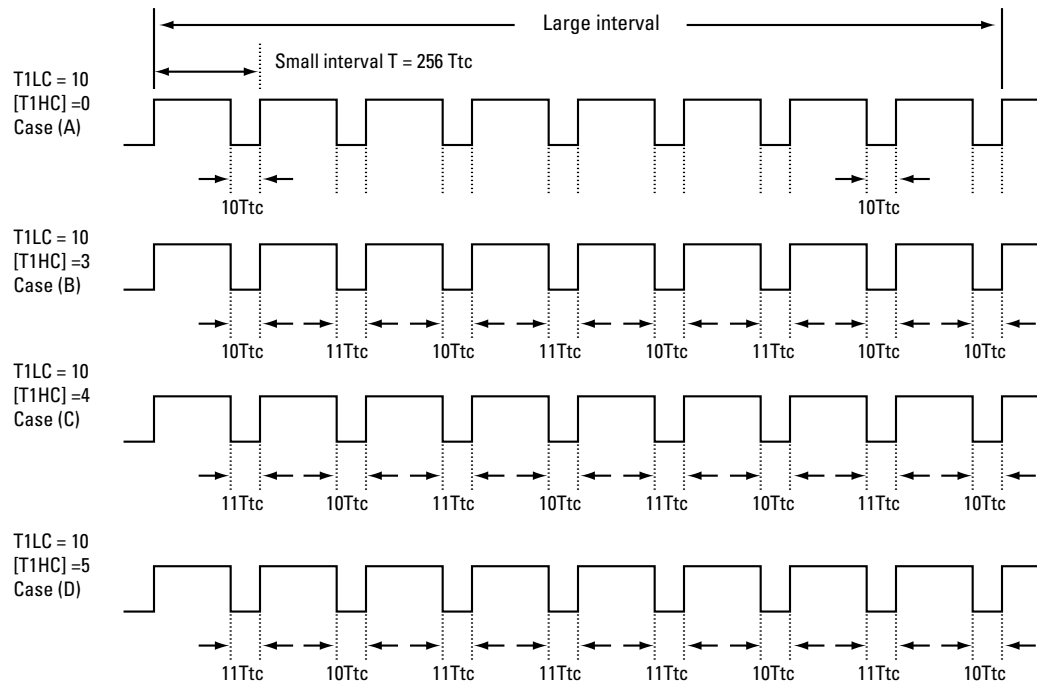
Next, consider the 11-bit pulse generator.

[BIT] = 11, large interval P cycle  $TP = 211 \times Ttc = 2048 Ttc$ . Since the small interval T is 256 Ttc (Ttc: pulse signal clock cycle), T is repeated 8 times in the large interval P.

Fig. 17-3 shows the change in output waveform caused by T1HC when T1LC is constant.

For [T1LC] = 10 (0AH), [T1HC] = 0, a "Low" level pulse of 10 Ttc is output for all small intervals.

For [T1HC] = 3, three small intervals with a "Low" level pulse width of 11 Ttc will be generated in the large interval, as shown by (B). For [T1HC] = 4, four small intervals with a "Low" level pulse width of 11 Ttc will be generated in the large interval, as shown by (C). For [T1HC] = 5, five small intervals with a "Low" level pulse width of 11 Ttc will be generated in the large interval, as shown by (D).



**Figure 2.103** T1HC Value and 11 Ttc "L" Level Pulse Count



# Symbol Table

**Caution:** The initial values are the values established by the BIOS after a reset.

Symbol	Address	R/W	Designation	Default value	See page
RAM (bank 0)	000H-0FFH	R/W	Data memory	XXXXXXXX (stored at reset)	43
RAM (bank 1)	000H-0FFH	R/W	Data memory	XXXXXXXX (stored at reset)	43
ACC	100H	R/W	Accumulator	00000000	50
PSW	101H	R/W	Program status word	00H00000	52
B	102H	R/W	B register	00000000	51
C	103H	R/W	C register	00000000	51
TRL	104H	R/W	Table reference register lower byte	00000000	54
TRH	105H	R/W	Table reference register upper byte	00000000	54
SP	106H	R/W	Stack pointer	XXXXXXXX	53
PCON	107H	R/W	Power control register	HHHHHH00	158
IE	108H	R/W	Master interrupt enable control register	0HHHHH00	138
IP	109H	R/W	Interrupt priority control register	00000000	151
EXT	10DH	R/W	External memory control register	HHHH0000	–
OCR	10EH	R/W	Oscillation control register	0H00HH00	156
TOCNT	110H	R/W	Timer 0 control register	00000000	67
TOPRR	111H	R/W	Timer 0 prescaler data	00000000	71

TOL	112H	R	Timer 0 low	00000000	71
TOLR	113H	R/W	Timer 0 low reload data	00000000	71
TOH	114H	R	Timer 0 high	00000000	72
TOHR	115H	R/W	Timer 0 high reload data	00000000	72
T1CNT	118H	R/W	Timer 1 control register	00000000	83
T1LC	11AH	R/W	Timer 1 low comparison data	00000000	86
T1L	11BH	R	Timer 1 low	00000000	85
T1LR		W	Timer 1 low reload data	00000000	85
T1HC	11CH	R/W	Timer 1 high comparison data	00000000	87
T1H	11DH	R	Timer 1 high	00000000	86
T1HR		W	Timer 1 high reload data	00000000	86
MCR	120H	W	Mode control register	00000000	127
STAD	122H	R/W	Start address register	00000000	129
CNR	123H	W	Character count register	H0000000	130
TDR	124H	W	Time division register	HH000000	130
XBNK	125H	R/W	Bank address register	HHHHHH00	130
VCCR	127H	W	LCD contrast control register	00000000	131
SCON0	130H	R/W	SIO0 control register	00H00000	108
SBUF0	131H	R/W	SIO0 buffer	00000000	113
SBR	132H	R/W	SIO0 baud rate generator	00000000	113
SCON1	134H	R/W	SIO1 control register	00000000	111
SBUF1	135H	R/W	SIO1 buffer	00000000	113
P1	144H	R/W	Port 1 latch	00000000	58
P1DDR	145H	W	Port 1 data direction register	00000000	58
P1FCR	146H	W	Port 1 function control register	10111111	59
P3DDR	14DH	W	Port 3 data direction register	00000000	62
P3INT	14EH	R/W	Port 3 interrupt function control register	11111101	62
P7	15CH	R	Port 7 latch	HHHHXXXX	64
I01CR	15DH	R/W	External interrupt 0, 1 control	00000000	135
I23CR	15EH	R/W	External interrupt 2, 3 control	00000000	137

ISL	15FH	R/W	Input signal select	11000000	138
VSEL	163H	R/W	Control register	11111100	143
VRMAD1	164H	R/W	System address register 1	00000000	144
VRMAD2	165H	R/W	System address register 2	HHHHHHH0	144
VTRBF	166H	R/W	Send/receive buffer	XXXXXXXX	144
BTCR	17FH	R/W	Base timer control	01000001	101
RAM (XRAM) (Bank 0)	180H-1FBH	R/W	LCD memory	XXXXXXXX (stored at reset)	126
RAM (XRAM) (Bank 1)	180H-1FBH	R/W			
RAM (XRAM) (Bank 2)	180H-185H	R/W			



# VMU Mode Selection

The operation modes available for the VMU and the selection principles are shown in the illustration below.

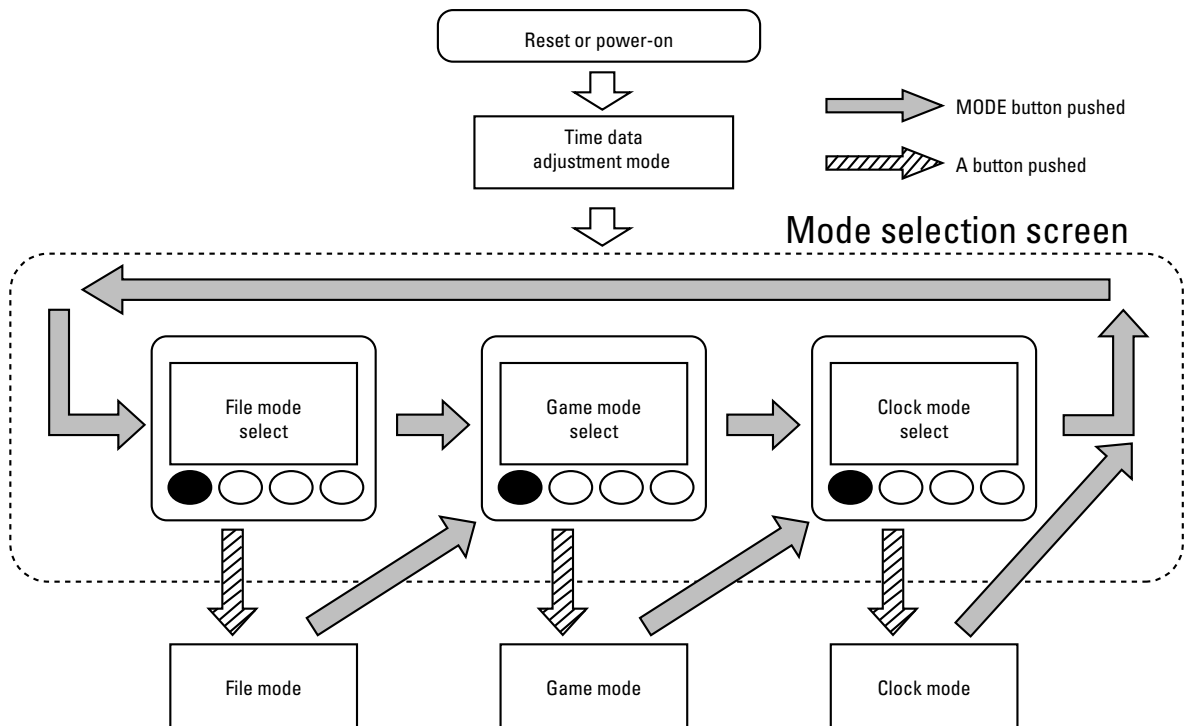


Figure 2.104 Mode Select Transition Diagram

## Mode selection screen

The mode selection screen serves to select and execute one of the three modes of the VMU.

With each push of the MODE button, the mode is switched. During selection, the corresponding LCD icon flashes. Pressing the A button then executes the selected mode.

### **File mode**

This mode serves for managing and editing game data and applications stored on the VMU. While the file mode is active, pressing the MODE button switches to the mode selection screen.

### **Game mode**

This mode serves for running an application transferred to the VMU.

The application must be programmed to restore the mode selection screen when the MODE button is pressed.

---

**Caution:** The return to the mode selection screen from the game mode is not supported by the BIOS. It must be incorporated in the application.

---

### **Clock mode**

In this mode, the current date and time are shown.

While the clock mode is active, pressing the MODE button switches to the mode selection screen. Keeping the A button depressed while pressing a direction key activates the time set mode.

---

# ***Calculation of Battery Life***

---

Because the VMU incorporates two system clocks with different current consumption, battery life will differ with different applications. Another important factor that influences battery life is whether a program is designed only for standalone operation or for use in conjunction with another VMU.

The instruction manual of the VMU therefore contains only the specification “With new lithium batteries, the built-in clock will operate continuously for about 130 days if only the OS is used.”

This section contains information about how to calculate expected battery life based on the source code of an application. Developers should use these data to determine expected battery life, and this information should be conveyed to the user. Since the price of a lithium battery of the type used in the VMU is approximately 280 Japanese yen (as of November 1998 in Japan), each replacement will cost the user about 560 yen for the two batteries. Programs should therefore be designed so as to consume as little power as possible.

## **Methods for Enhancing Battery Life**

The VMU incorporates two system clocks, an RC oscillator (879.236 kHz; tolerance range 600 to 1200 kHz), and a quartz oscillator (32.768 kHz).

The RC oscillator increases processing speed compared to the quartz oscillator, but it also consumes more power. An important consideration when programming an application is therefore how to use the RC oscillator as little as possible without impairing playability.

Writing to XRAM or flash memory always requires use of the RC oscillator, which will increase the RC oscillator load. When two VMU units are connected for serial transfer, the load will also increase. Depending on the circuit configuration, not only the transfer but also the connection process itself can consume considerable power.

With regard to power consumption, take the following points into account when coding an application.

- Use the RC oscillator as little as possible.
- Avoid frequent writing accesses to flash memory.
- Keep communication sessions short.
- Clearly specify the connect/disconnect timing for communication applications.
- Do not redraw the LCD frequently.

## Oscillator Circuit and Current Consumption

The following table shows the current consumption of the two oscillator circuits.

Oscillator circuit	Clock frequency	Current consumption
RC oscillator	879.236 kHz	2.600 mA
Quartz oscillator	32.768 kHz	0.610 mA

---

**Caution:** An RC oscillator inherently is subject to frequency tolerances. The oscillator used in the VMU has a tolerance range of 600 - 1200 kHz. Consequently, there will also be differences in current consumption. The battery life calculations in this section assume a frequency of 1000 kHz. The tolerance of the quartz oscillator is 50 ppm - 30 ppm from the center frequency of 32.768 kHz.

---

When using the RC oscillator, select a cycle time of 1/12 the system clock, except when writing to the flash memory.

## Oscillation Control Register

The oscillation control register (OCR) serves for selecting the oscillator circuit, start/stop control, and setting the system clock division ratio. By effectively managing these settings, battery life can be extended.

### System Clock Division Ratio Setting

The OCR7 bit serves for setting the cycle time to 1/12 or 1/6 of the system clock.

When set to "1", the cycle time is 1/12 of the system clock. For drawing the LCD image, use the RC oscillator with the 1/12 setting.

When reset to "0", the cycle time is 1/6 of the system clock. At this setting, current consumption is about 1.2 times higher than at the 1/12 setting. For writing to the flash memory, use the RC oscillator with the 1/6 setting.

Division ratio	Quartz oscillator cycle time	RC oscillator cycle time
1/12 (OCR7 = 0)	366.210 ms	12.568 ms
1/6 (OCR7 = 1)	183.105 ms	6.284 ms

## Oscillator Circuit Selection

The OCR5 and OCR4 bits serve for selecting the oscillator circuit.

When set to OCR5 = 0, OCR4 = 0, the RC oscillator is used for the system clock.

When set to OCR5 = 1, OCR4 = 0, the quartz oscillator is used for the system clock.

---

**Caution:** OCR5 and OCR4 only select the oscillator circuit to be used for the system clock. To reduce power consumption, it is also necessary to perform start/stop control of the RC oscillator.

---

When switching the system clock to the stopped oscillator circuit, insert a wait of at least 300 microseconds.



### Oscillator Circuit Start/Stop

The OCR1 bit serves for starting and stopping the RC oscillator.

When set to "1", the RC oscillator is stopped.

When reset to "0", the RC oscillator starts or continues to operate.

The RC oscillator should be stopped while using the quartz oscillator. When the RC oscillator operates, current consumption is about 1.1 times higher.

---

**Caution:** The quartz oscillator is used by the clock and therefore should not be stopped.

---

### Calculating Battery Life

Use the "Battery Life Calculation Chart" at the end of this section to calculate expected battery life, as follows.

The lithium battery (CR2032) used in the VMU has a capacity of 210 mAh. 82% (174 mAh) of this can be used by the VMU.

Before performing the calculation, the source code of the application under development must be available. Main processing parts should be extracted as model programs.

Calculate total number of instruction cycles where quartz oscillator is operating

Enter this value at position A in the chart.

Calculate total number of instruction cycles where RC oscillator is operating

Enter this value at position B in the chart.

---

**Reference:** For information on instruction cycles used by the various instructions, refer to the "VMU Programmer's Guide".

---

### Calculating Continuous Operating Time

#### Calculate division ratio

Calculate RC oscillator operation time at 1/12 ratio setting (OCR7 = 0) and 1/6 setting (OCR7 = 1)

Enter these values at positions C and D in the chart.

#### Calculate total operation time of quartz oscillator

There are two C positions in the chart. Add up the two values, multiply by 30.5 ms, and enter the result at position E in the chart.

#### Calculate total operation time of RC oscillator

There are two D positions in the chart. Add up the two values, multiply by 1 ms, and enter the result at position F in the chart.

### Calculate average current consumption of quartz oscillator

Multiply the value in E by 0.610 mA, and enter the result at position G in the chart.

### Calculate average current consumption of RC oscillator

Multiply the value in F by 2.600 mA, and enter the result at position H in the chart.

### Calculate combined average current consumption of quartz oscillator and RC oscillator

Calculate  $(G + H) \div (E + F)$ , and enter the result at position I in the chart.

Calculate battery life (hours) from effective battery capacity (190 mA)

174 mAh  $\div$  I yields battery life J (hours).

### Provide a 10% safety margin.

Possible factors influencing the actual battery life are sub-processing cycles, flash memory write access, load during data transfer, temperature influences, etc.

$J \times 0.9 = K$  (battery life in hours, with margin)

This value can be included in product documentation. It should be defined as the expected life of one set of batteries for continuous operation of the software.

## Calculating Battery Life in Days

The following steps show how to calculate expected battery life in days, assuming a certain number of hours of use every day, and assuming that the unit is in sleep mode (0.060 mA) at other times.

### Calculate current consumption during play hours

Calculate  $I \times$  number of play hours per day L (hours), and enter the result at position M in the chart.

### Calculate daily current consumption in sleep mode

Calculate  $0.060 \text{ mA} \times (24 \text{ hours} - L)$ , and enter the result at position N in the chart.

### Calculate daily average current consumption

Calculate  $(M + N) \div 24$ , and enter the result at position O in the chart.

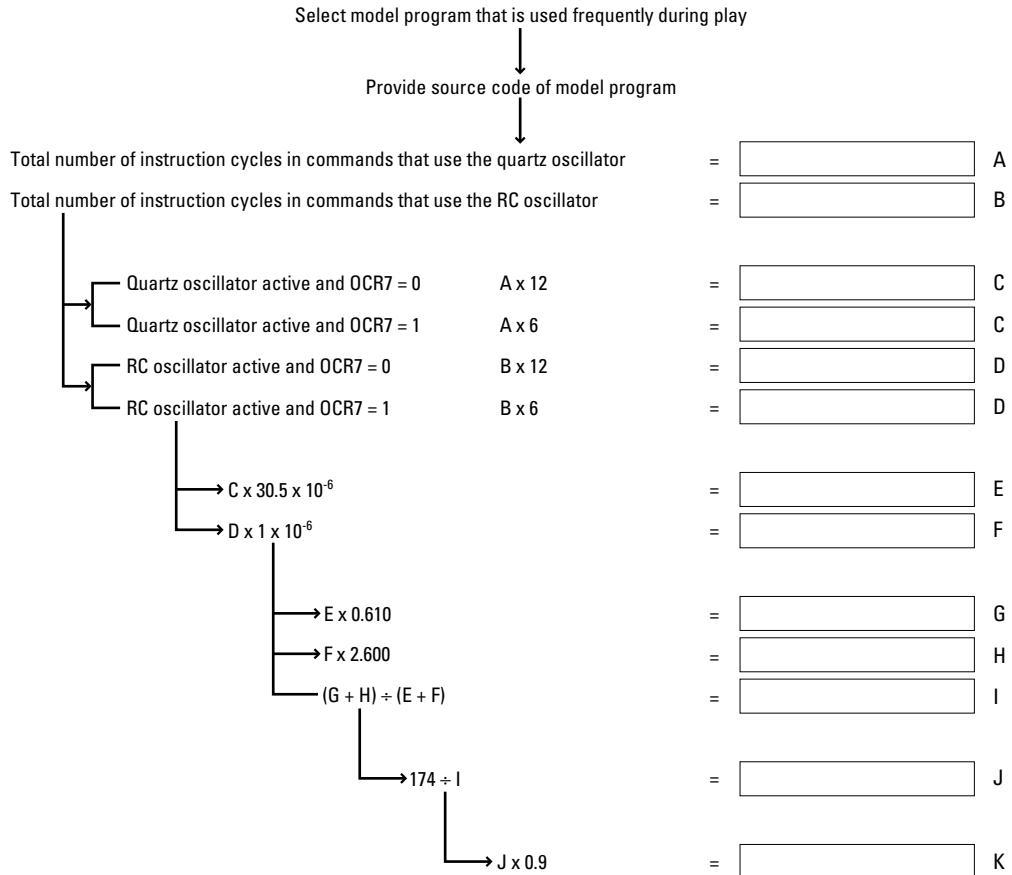
### Calculate effective battery life in days

Calculate  $(174 \div 24) \div O$ , and enter the result at position O in the chart.

### Provide a 10% safety margin.

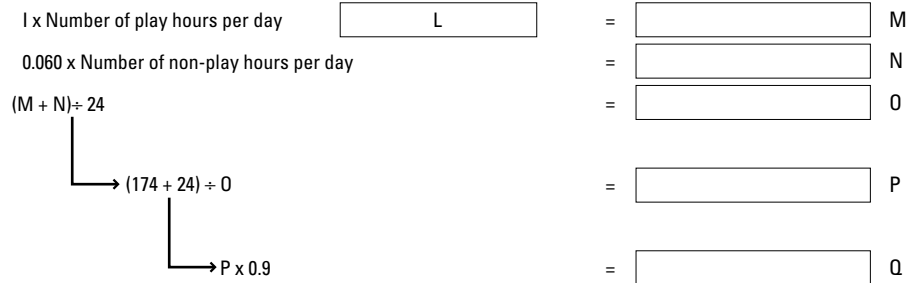
$P \times 0.9 = Q$  (battery life in days, with margin)

This value can be included in product documentation. It should be defined as the expected life of one set of batteries when using the software for n hours per day.



The expected life of one set of batteries when operating the software continuously is  hours.

Expressing battery life in days



The expected life of one set of batteries when operating the software for  hours per day is  days.

**Figure 2.105**



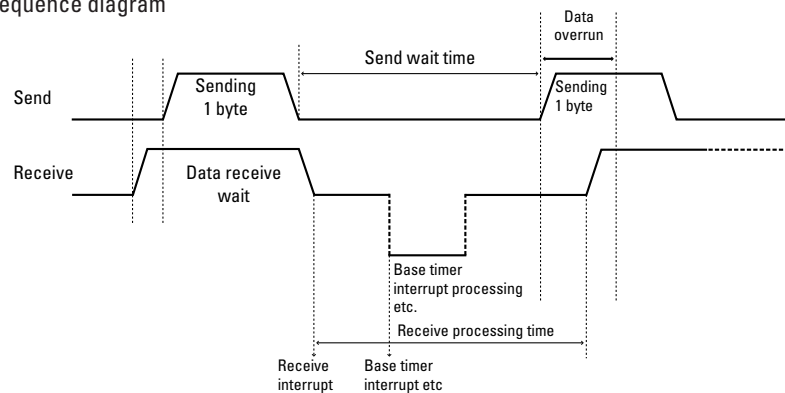
# Serial Communication Precautions

This chapter describes points to be observed regarding serial communication between two connected VMU units. Techniques for ensuring proper communication are also explained.

## Serial Communication Timing Chart

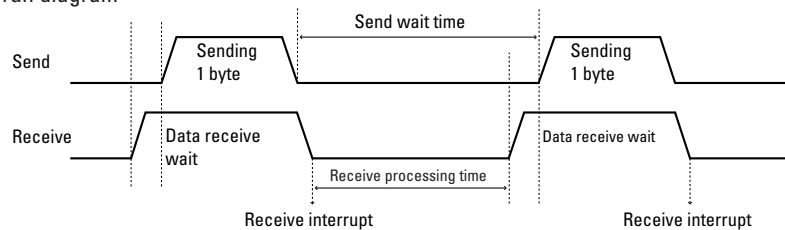
A timing chart for connecting two VMU units and performing serial communication is shown below.

Correct sequence diagram



When data are sent while the receive side is in the receive wait condition, the data will be received correctly. Otherwise, data overrun may occur, as shown in the next illustration.

Data overrun diagram



When the send wait time is longer than the receive processing time, data transfer will be carried out correctly. However, the base timer interrupt is generated also during receive processing, and the processing time will be longer than the actual time required for all receive processing handler steps.

## Measures to Ensure Problem-Free Serial Transfer

The following measures for ensuring smooth serial transfer are possible.

- 1) Mask all interrupts except those needed for serial transfer.
- 2) Give highest priority to receive interrupt.
- 3) Make send wait time longer than sum of receive wait time and time required for other processing steps.

Method (1) will result in a slow-down of the built-in clock of the VMU, because the clock uses the base timer interrupt, and interrupts cannot be counted while masking is active.

If method (2) were adopted, the receive interrupt handler located in the flash memory space would not be called, because the base timer interrupt handler (clock processing routine) is located in the ROM space.

When a base timer interrupt is generated, the CHANGE instruction causes the CPU to start program fetch from ROM. When a receive interrupt occurs while the base timer interrupt of the ROM BIOS is being processed, the CPU references the interrupt table (equivalent address) in the ROM space and jumps to that address. Because the CPU is carrying out ROM fetch, jumping to the appropriate receive interrupt handler is not possible. Therefore method (2) is not suitable and cannot be used.

For method (3), the processing time required for all interrupts with higher priority than the receive interrupt is calculated and added to the receive processing time, and the result is taken as the send wait time. This ensures correct data transfer but increases the time required for sending.

The decision of whether to adopt method (1) or (3) must be made by the application designer, while taking into account the advantages and disadvantages of either approach. These are described in more detail below.

### Mask All Interrupts

When all interrupts except those needed for serial transfer are masked, the clock which uses the base timer interrupt will be slowed down. But compared to method (3), method (1) will result in higher transfer speed and simplified programming logic.

For transfer of only a few bytes, the clock slow-down will be within the tolerance specifications. But for frequent data transfer or transfer of a larger amount of data, method (3) should be adopted.

The send wait time is calculated as follows.

- 1) Count the total number of steps required for receive processing.

In addition to the number of steps in the interrupt processing handler, also include the steps immediately before and immediately after the interrupt.

- 2) Check the system clock and division ratio used during execution of these steps.
- 3) Calculate the send wait time from the total number of steps, system clock, and division ratio.
- 4) Incorporate the wait time calculated in (3) into the NOP or similar for send side processing.

## Set Maximum Send Wait Time

Compared to masking all interrupts, this method results in a considerable decrease of transfer speed. When coding an application using this method, the send wait time calculation and priority assignment must be carried out with care. The advantage of this method is that there will be no clock slow-down.

For calculating the send wait time, it is assumed that all interrupts with higher priority than the receive interrupt have been generated. The processing time required for these is added to the receive processing time, and the result is taken as the send wait time.

---

**Note:** Because the port 3 interrupt is a level interrupt, it will be generated continuously for as long as the user presses a button. If this happens during a transfer, data overrun may occur regardless of how long the send wait time is set.

---

For checking the button press status during a serial transfer, inhibit the port 3 interrupt and use timer 0 or other means for monitoring port 3 latch data.

- 1) Check whether there is an interrupt with equal priority to the receive interrupt. If there is such an interrupt, set the interrupt to a higher or lower priority than the receive interrupt.
- 2) Count the total number of steps required for receive processing. In addition to the number of steps in the interrupt processing handler, also include the steps immediately before and immediately after the interrupt.
- 3) Check the system clock and division ratio used during execution of these steps.
- 4) Calculate the processing time from the total number of steps, the system clock and division ratio.
- 5) Pick up interrupts with higher priority than the receive interrupt, except the base timer interrupt.
- 6) Count the total number of steps for the interrupt handlers of these interrupts.
- 7) Calculate the processing time for each handler, according to the method of steps (2) to (4).
- 8) Calculate the base timer interrupt processing time. Because the base timer interrupt handler is processed within the BIOS, the number of steps can be assumed as shown below. These figures apply only if GHEAD.ASM was not changed. If GHEAD.ASM was changed, count the actual number of steps.

### Shortest case

Second count processing only

	5 steps (until CHANGE instruction in GHEAD.ASM)
+	48 steps (in BIOS)
+	3 steps (until return to user program in GHEAD.ASM)
Total	56 steps

### Longest case

Year increment processing necessary

5 steps (until CHANGE instruction in GHEAD.ASM)

+ 145 steps (in BIOS)

+ 3 steps (until return to user program in GHEAD.ASM)

Total 153 steps

---

**Caution:** Because the longest case shown above will occur only once per year, it is not necessary to always provide for this number of steps.  
The system clock and division ratio are derived from the settings established by the application.

---

- 9) Calculate the base timer interrupt processing time from the system clock and division ratio.
- 10) Add up the receive interrupt handler processing time and the time required for all calculated interrupts.
- 11) Incorporate the wait time calculated in (10) into the NOP or similar for send side processing.



***Visual Memory Unit (VMU)  
Programming Manual***



# ***Table of Contents***

<b>Setup</b> .....	<b>VMC-1</b>
Executing the Setup Program .....	VMC-1
Post-Installation Overview .....	VMC-7
 <b>Setting Environment Variables</b> .....	 <b>VMC-9</b>
Environment Variables for the Development Tools .....	VMC-9
Environment Variable Settings .....	VMC-10
 <b>Specifying Files for Assembly</b> .....	 <b>VMC-11</b>
Specifying File Names .....	VMC-11
Specifying Parameters on the Command Line .....	VMC-12
Specifying Parameters at the Prompts .....	VMC-13
 <b>Option Switches</b> .....	 <b>VMC-15</b>
 <b>Environment Variables and Reserved Word File</b> .....	 <b>VMC-17</b>
Environment Variables .....	VMC-18
Reserved Word File .....	VMC-19
 <b>Errors</b> .....	 <b>VMC-21</b>
Warnings .....	VMC-22
Non-Fatal Errors .....	VMC-25
Fatal Errors .....	VMC-31

<b>Listing Format</b> .....	<b>VMC-35</b>
-----------------------------	---------------

<b>Specifying Files for Linking</b> .....	<b>VMC-39</b>
---	---------------

Specifying File Names .....	VMC-40
Specifying Parameters on the Command Line .....	VMC-41
Specifying Parameters at the Prompts .....	VMC-42
Files Referenced During Linking .....	VMC-44

<b>Option Switches</b> .....	<b>VMC-45</b>
------------------------------	---------------

<b>Object Alignment</b> .....	<b>VMC-49</b>
-------------------------------	---------------

-A option .....	VMC-50
-A -F options .....	VMC-51
-A -O options .....	VMC-52
-A -R options .....	VMC-53

<b>Errors</b> .....	<b>VMC-55</b>
---------------------	---------------

Fatal Errors .....	VMC-55
Non-Fatal Errors .....	VMC-56

<b>Starting the Program</b> .....	<b>VMC-57</b>
-----------------------------------	---------------

Specifying File Names .....	VMC-57
Specifying Parameters on the Command Line .....	VMC-58
Option .....	VMC-59
Examples of Command Line Execution .....	VMC-59
Operation with the Prompts .....	VMC-60
Prompt Line Extension .....	VMC-60
Default Responses .....	VMC-60

<b>Error Messages</b> .....	<b>VMC-61</b>
-----------------------------	---------------

<b>Cross-Reference</b> .....	<b>VMC-63</b>
------------------------------	---------------

<b>Starting the Program</b> .....	<b>VMC-65</b>
-----------------------------------	---------------

Specifying File Names .....	VMC-66
Specifying Parameters .....	VMC-67
Option Specification .....	VMC-68

<b>Error Messages</b> .....	<b>VMC-69</b>
-----------------------------	---------------

Fatal Errors .....	VMC-69
--------------------	--------

<b>Starting the Program</b> .....	<b>VMC-71</b>
Specifying File Names .....	VMC-71
Specifying Parameters .....	VMC-72
<b>Error Messages</b> .....	<b>VMC-73</b>
Fatal Errors .....	VMC-73
<b>Overview of MAKE</b> .....	<b>VMC-75</b>
Running MAKE .....	VMC-76
Build Priority Sequence .....	VMC-76
Command Line Options .....	VMC-76
Makefile Syntax .....	VMC-78
Generation Rules .....	VMC-78
Macros .....	VMC-80
Directives .....	VMC-81
Implicit Rules .....	VMC-82
Makerule file .....	VMC-82
<b>Assembler Syntax</b> .....	<b>VMC-85</b>
Statements .....	VMC-85
Label and Symbol Names .....	VMC-86
Comments .....	VMC-86
Operators .....	VMC-86
Numeric Constants .....	VMC-87
Character Constants .....	VMC-88
Character String Constants .....	VMC-89
Special Symbols .....	VMC-89
<b>Assembler Pseudoinstructions</b> .....	<b>VMC-91</b>
<b>LC86K Instruction Summary</b> .....	<b>VMC-147</b>
Instruction Summary .....	VMC-147
Arithmetic Instructions .....	VMC-147
Logical Instructions .....	VMC-148
Data Transfer Instructions .....	VMC-148
Jump Instruction .....	VMC-148
Conditional Branch Instructions .....	VMC-149
Subroutine Instruction .....	VMC-149
Bit Manipulation Instructions .....	VMC-149
Other Instructions .....	VMC-149
Macro Instruction .....	VMC-149
Addressing .....	VMC-149
Program Memory Addressing .....	VMC-150
RAM and Special Function Register (SFR) Addressing .....	VMC-152

**Instruction Set Reference ..... VMC-155**

Arithmetic Instructions ..... VMC-156  
    Logical Instructions ..... VMC-173  
    Data Transfer Instructions ..... VMC-186  
Jump Instructions ..... VMC-197  
Conditional Branch Instructions ..... VMC-201  
Subroutine Instructions ..... VMC-214  
Bit Manipulation Instructions ..... VMC-219  
Miscellaneous Instruction ..... VMC-222  
Macro Instruction ..... VMC-223

**LC86K Instruction Set Summary ..... VMC-225**

**Assembler Pseudoinstructions ..... VMC-227**

# Setup

---

This installs the Visual Memory tools supplied with the Dreamcast SDK.

**Caution:** This description assumes Version 1.3J of the Dreamcast SDK.

---

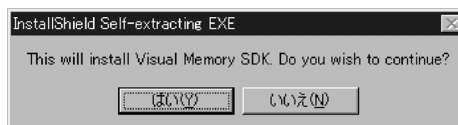
## Executing the Setup Program

- 1) Insert the Dreamcast SDK CD-ROM into the drive, and using Explorer or another tool, open the VM\_SDK directory.

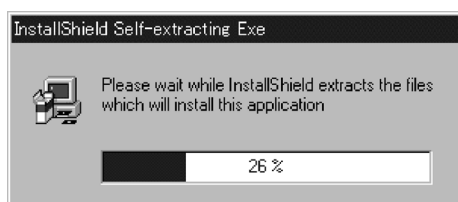
**Caution:** There may be last-minute changes which do not appear in the manual. Be sure to check the README.TXT file.

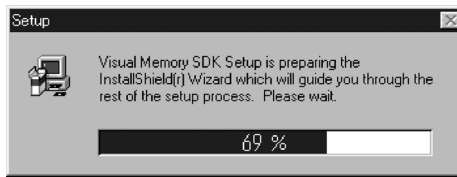
---

- 2) For automatic setup, including environment variables, double-click on VMSETUP2.EXE.
- 3) When the next dialog box appears, click the Yes button.

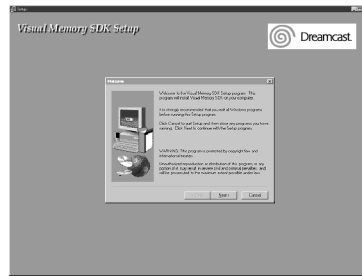


- 4) When the next dialog box appears, wait until the progress indicator reaches 100%. Two dialog boxes are shown.

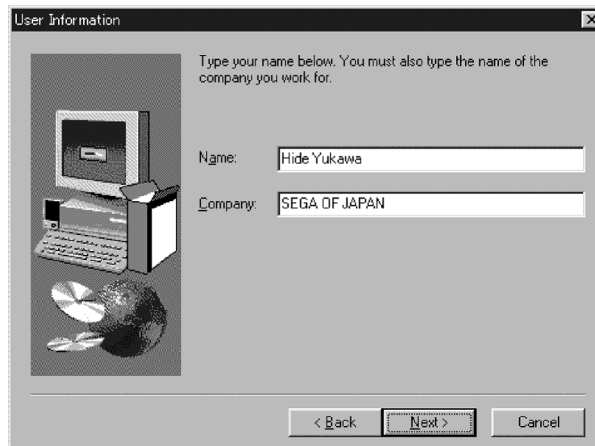




5) The installer has now started. Click the Next button in the dialog box.

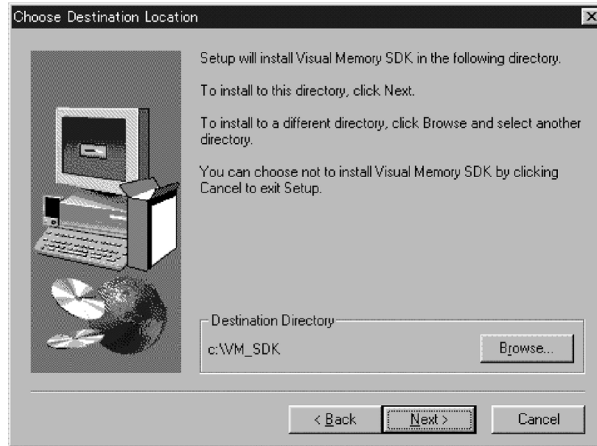


6) When the next dialog box appears, enter your name and your company name. When these are complete, click Next.

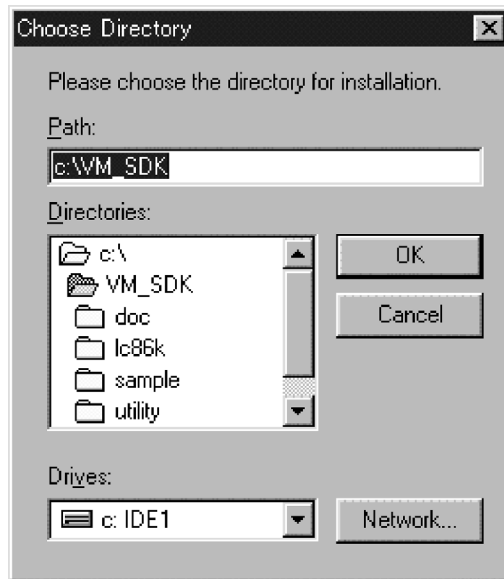


7) When the next dialog box appears, enter the drive and path specification of where you want to install the Visual Memory SDK. After you have entered this, click Next.





To install with a different path specification, click Browse, and enter the drive and path specification.



8) When the next dialog box appears, select the internal components of the Visual Memory SDK to be installed.



Select **Typical** to install all SDK components.

Select **Compact** to install the whole SDK except for sample programs.

Select **Custom** to select individual components in the next dialog box.

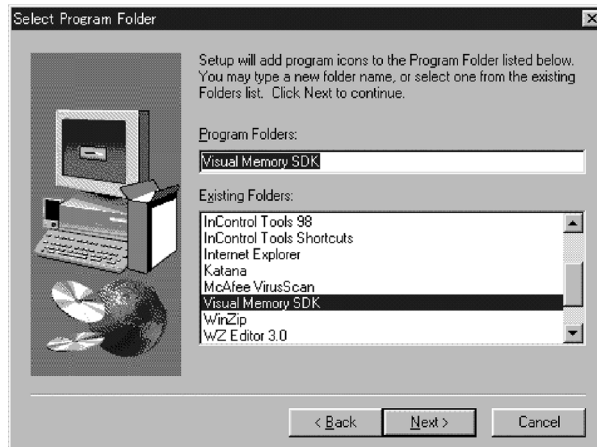


Select **Visual Memory SDK** to install the assembler, linker, and other tools, and the Memory Card Utility for application transfer.

Select **Visual Memory Simulator** to install the Visual Memory simulator.

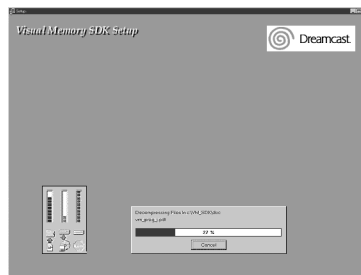
Select **Sample Programs** to install the sample programs.

9) Add the program to the start menu. Enter the name, then click Next.



**Caution:** Only the Visual Memory simulator can be added to the Start menu. Other tools are MS-DOS programs, and cannot be added to the Start menu.

10) Copying of the files now starts. Wait until this finishes.



11) A restart confirmation dialog box appears. Select **Yes, I want to restart my computer now** and click **Finish** to restart the computer immediately.



12) If the following dialog box appears, click Next.



---

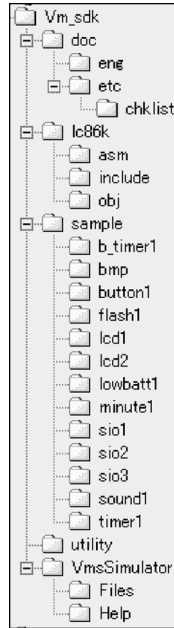
**Note:** The setup program adds the following to the AUTOEXEC.BAT file.

```
SET PATH=%PATH%;C:\VM_SDK\LC86K
SET CHIPNAME=LC868700
SET M86KRSVDFILE=c:\VM_SDK\lc86k\m86krsvd.rwd
SET TOOL86=c:\VM_SDK\lc86k
```

---

## Post-Installation Overview

Installing the Visual Memory SDK creates the following directories.



Name	Meaning
Utility	Contains the utility for transfers PC_Dev. Box_Visual Memory, This program is a Dev. Box executable file. Run it using CodeScape or similar.
Doc	Contains documents relating to Visual Memory.
lc86k	Contains the assembler, linker, and other development tools. Also includes header files, libraries, and the GHEAD.ASM file.
Sample	Contains sample programs.
VMUSimulator	Contains the Visual Memory simulator.



# Setting Environment Variables

## Environment Variables for the Development Tools

The following environment variables are used with the L86K series development tools.

### PATH

Defines the search path, by adding to the existing PATH definition.

Name	Search file
M86K	Searches for the reserved word file M86KRSVD.RWD in the directories specified by the PATH variable.
L86K	Searches for the reserved word defined symbol file LC86L.LIB in the directories specified by the PATH variable.
CGR86K	Searches for the default character generator data file DEFAULT.CGR in the directories specified by the PATH variable.

### CHIPNAME

This defines the chip name (or series name) on which the software will run.

For developing Visual Memory applications, specify LC868700.

Name	Meaning
M86K	Defines the chip name for the assembly. This is ignored if there is a CHIP pseudoinstruction (see Chapter 21, "Assembler Pseudoinstructions") in the source program; it is referenced at assemble time if there is no CHIP pseudoinstruction. If the ROM size of the chip (the last two digits) is set to "00", then no ROM size check is carried out, and the assembly proceeds as though with 64K installed.
SU86K	Defines the chip name for option data creation. The ROM size specification (the last two digits of the chip name) is ignored.
CGR86K	Defines the chip name for character generator data file creation. The ROM size specification (the last two digits of the chip name) is ignored.

If the last two digits of the chip name are "00", this does not indicate a specific chip. It is a generic label for a series, but is given the maximum ROM capacity in the series. Such a chip name can be used in order to determine the size of an assembled program.

### M86KRSVDFILE

Defines the name of the reserved word file.

Name	Search file
M86K	Defines the directory in which the reserved word file is stored, and the file name. If this environment variable is not defined, the default file name is M86KRSVD.RWD, and the directories in the PATH specification are searched in sequence for the file M86KRSVD.RWD.

### M86KWORKFILE

Defines the name of the work file.

Name	Search file
M86K	Specifies the name of the file which will be used as a kind of expansion memory for work space when the working memory dynamically allocated during the assembly process of the M86K compiler will not fit into main memory, and EMS memory is not installed, or is not available. It is possible to specify a drive and path name.

### TOOL86K

Defines the directory holding the assembler, linker, LC86K.LIB, and so on.

---

**Caution:** This environment variable must be set.

---

### TMP

Defines the directory for temporary workfiles.

Name	Search file
M86K	When M86K needs to create a work file (see "M86KWORKFILE"), if the environment variable M86KWORKFILE is not defined, then a work file is created in the directory specified by this environment variable. If this environment variable is not defined, the work file is created in the current directory. In either case, the file name is fixed, and is always M86KWORK.TMP.

## Environment Variable Settings

Use the SET command to set environment variables in MS-DOS. For details of the SET command, refer to the documentation and help files supplied with MS-DOS.

### Example: To set the default chip name to LC868700

```
C> SET CHIPNAME=LC868700
```



# Specifying Files for Assembly

There are two ways of specifying the parameters when starting M86K.

- 1) Specifying all parameters on the M86K command line
- 2) Specifying parameters by responding to the prompts displayed by M86K

To force an exit from M86K, press the following key combinations.

Computer type	Keys
PC/AT compatible	Ctrl + C or Ctrl + Pause/Break
NEC PC-9800 series	CTRL + C or Ctrl + Stop

## Specifying File Names

When specifying file names to M86K, either in the command line, or in response to the prompts, case is ignored. For example, the following file names all refer to the same file.

```
sample.asm
SAmpLE.ASM
SAMPLE.asM
```

If a file name is specified without an extension, M86K supplies the following default extensions.

File format	Default extension
Source file	.ASM
Object file	.OBJ
Listing file	.LST
Cross-reference file	.CRF
Error file	.ERR

## Specifying Parameters on the Command Line

M86K\_options\_ \_msource\_ \_mobject\_ \_mlist\_ \_mcross\_ \_merror\_ ↵

### **options**

Specify assembler options as listed in Chapter 4, “Option Switches.” If options are specified they must come before other parameters.

### **source**

Specify the source file to be assembled. If the file extension is omitted, the default extension .ASM is added. If a file extension is specified, it takes precedence. In either case, a drive name and full path specification is possible.

### **object**

Specify the object file name for the assembler output. A drive name and full path specification is possible. If this file specification is omitted, the object file name is the source file name, with the extension changed to .OBJ. If the file already exists, it is overwritten.

### **list**

Specify a file name for the assembly listing output. A drive name and full path specification is possible. If this file specification is omitted, no listing file is created. If the listing file already exists, it is overwritten.

### **cross**

Specify a file name for the assembly symbol cross-reference output. A drive name and full path specification is possible. If this file specification is omitted, no cross-reference is output. If the cross-reference file already exists, it is overwritten.

### **error**

Specify a file name for the output of assembly error messages. A drive name and full path specification is possible. If this file specification is omitted, no error file is created. If the error file already exists, it is overwritten.

### **Example**

```
C> M86K MAIN.ASM,MAIN, ,TEST.CXX ø
```

This begins assembly of the MAIN.ASM source file in the current directory. The object file MAIN.OBJ is created; no listing file is output, but a cross-reference is written to file TEST.CXX,

## Specifying Parameters at the Prompts

Enter the following command to start the assembler without specifying any file names. Then follow the prompts from the assembler to enter the file names.

```
prompt M86K [options]Ø
SANYO (R) LC86K series Macro Assembler Version X.XX
Copyright (c) SANYO Electric Co., Ltd. 1989-1995. All rights reserved.
Source filename[.ASM]:
Object filename[.OBJ]:
Source listing [NUL.LST]:
Cross reference[NUL.CRF]:
Error messages [NUL.ERR]:
```

### options

Specify assembler options as listed in Chapter 4, "Option Switches."

### Source filename

Specify the source file to be assembled. If the file extension is omitted, the default extension .ASM is added. If a file extension is specified, it takes precedence. In either case, a drive name and full path specification is possible. This file name may not be omitted. Pressing the Enter key alone results in a reprompt for the source file name. To cancel the operation at this point, use the following keys.

Computer type	Keys
PC/AT compatible	Ctrl + C or Ctrl + Pause/Break
NEC PC-9800 series	Ctrl + C or Ctrl + Stop

### Object filename

Specify the object file name for the assembler output. A drive name and full path specification is possible. If this file specification is omitted (by pressing the Enter key alone), the object file name is the source file name, with the extension changed to .OBJ. If the file already exists, it is overwritten.

### Source listing

Specify a file name for the assembly listing output. A drive name and full path specification is possible. If this file specification is omitted (by pressing the Enter key alone), no listing file is created. If the listing file already exists, it is overwritten.

### Cross reference

Specify a file name for the assembly symbol cross-reference output. A drive name and full path specification is possible. If this file specification is omitted (by pressing the Enter key alone), no cross-reference is output. If the cross-reference file already exists, it is overwritten.

### Error messages

Specify a file name for the output of assembly error messages. A drive name and full path specification is possible. If this file specification is omitted (by pressing the Enter key alone), no error file is created. If the error file already exists, it is overwritten.



---

# Option Switches

---

This chapter describes the assembler options which can be used to control the operation of M86K itself. All options are indicated by a prefixed minus sign or slash. The options are not case dependent. Thus, for example, `-i`, `-I`, `/i`, and `/I` all mean the same.

## **-I**

### **Do not distinguish case in identifiers**

When this switch is specified, the assembler ignores case in treating user-defined identifiers (labels, macro names, and symbols). If this switch is not specified, then uppercase and lowercase letters are distinguished. This applies only to user-defined identifiers, and not to mnemonics, SFR names and so forth.

## **-D**

### **Do not output debugging information**

When this switch is specified, the assembler does not include symbol information and source line numbers in the object file. If this information is not present in the object file, source line mode debugging is not possible. If this switch is not specified, symbol information and source line numbers are both included, and source line mode debugging is possible.

## **-J**

### **Do not optimize jump instructions**

If this switch is specified when assembling a source including pseudoinstructions (JMPO, CALLO, BRO) which require optimization, the optimization is suppressed. The result is that each of these pseudoinstructions is interpreted as a three-byte instruction, regardless of the jump destination. If this switch is not specified when assembling a source including pseudoinstructions which require optimization, then the optimization is carried out. If the source does not include pseudoinstructions which require optimization, this switch has no effect.

**-N****Suppress copyright notice**

When this switch is specified, the copyright notice displayed when the assembler starts up is suppressed. This is convenient when starting the assembler from make or another utility, so that error messages are not obscured by other unnecessary messages.

**-R****Specify reserved word file**

The character string immediately following this switch until the next space indicates the name of the reserved word file to the assembler. The following example shows this:

```
m86k -rm86krsvd.rwd source.asm, ,source.lst
```

In this example the name of the reserved word file name is M86KRSVD.RWD. This specification takes precedence over a setting of the M86KRSVDFILE environment variable.

**-P****Specify working buffer size**

The numerical value immediately following this switch is used as the size of the assembler's internal working buffer. The working buffer is a memory area used by the assembler for increasing the speed of macro registration and expansion, and is reserved in main memory when the assembler is started. The default size is 4096 bytes, and this is normally sufficient for most source programs. If the assembler runs out of working buffer, however, the following error message is issued.

```
no more PARAMETER buffer (123) 45
```

The assembler issues this message, then abandons processing (the two digits at the end of the message are an internal value, and may change). If this message should appear, use this switch to specify a large buffer size, and repeat the assembly. For example, use the following specification:

```
m86k -p8192 source.asm
```

In this case, the working buffer size is set to 8192 bytes. The value must be in decimal, and immediately follow the switch letter 'p'. If the switch letter only is specified, or the numeric value cannot be found, the buffer size reverts to the default 4096 bytes.

**-?****Show options**

If this switch is specified, the assembler displays the following list of options, then immediately terminates. Note that any other option specifications are ignored.

```
Usage: m86k [option] source,[object],[list],[xref]
```

```
option:
```

```
  /D          do not make local symbol table and source line  
              attributes in object file  
  /I          ignore case for user defined symbol  
  /J          do not try to optimize  
  /N          skip displaying copyright message  
  /Psize     parameter buffer size in decimal  
  /Rfile     read `file' as reserved word file
```

# ***Environment Variables and Reserved Word File***

---

## Environment Variables

In MS-DOS, to set environment variables use the SET command. For details of the SET command, refer to your MS-DOS documentation and help files.

**Example: set the default chip name to LC868700.**

```
C> SET CHIPNAME=LC868700ø
```

M86K refers to the following environment variables as required.

### **PATH**

This is used as the search path for the reserved word file. For details of the reserved word file and the search algorithm used to find it, see Section , "Reserved Word File,".

### **CHIPNAME**

This defined the chip for which assembly will be carried out. It is ignored if the CHIP pseudoinstruction appears in the source program. However, if the chip name in the CHIP pseudoinstruction is different from the environment variable chip name, a warning message is issued. This value is thus used if the CHIP pseudoinstruction does not appear in the source program.

### **M86KRSVDFILE**

This defines the directory in which the reserved word file is stored, and the file name. Note that there is no default extension for the file name defined by this environment variable. Specify the drive name and path as required, but always specify the file name and extension.

### **M86KWORKFILE**

Specifies the name of the file which will be used as a kind of expansion memory for work space when the working memory dynamically allocated during the assembly process of the M86K assembler will not fit into main memory, and EMS memory is not installed, or is not available. It is possible to specify a drive and path name.

### **TMP**

When M86K needs to create a work file (see "M86KWORKFILE"), if the environment variable M86KWORKFILE is not defined, then a work file is created in the directory specified by this environment variable. If this environment variable is not defined, the work file is created in the current directory. In either case, the file name is fixed, and is always M86KWORK.TMP.



## Reserved Word File

The reserved word file is always read when M86K starts up, and includes various information specific to the chip for which the program will be assembled (RAM/ROM size, SFR mnemonics, etc.). M86K will not operate correctly without the reserved word file. When M86K starts, it searches for the reserved word file in the following sequence.

- 1) When a file name is explicitly specified by assembler option-R, that file is loaded. If the file is not present or reads are inhibited, an error occurs.
- 2) If the environment variable M86KRSVDFILE is defined, the file specified in that variable is loaded. If the file is not present or a read-only file, an error occurs.
- 3) If reads are enabled and the file M86KRSVD.RWD is present in the directory containing M86K.EXE, that file is loaded.
- 4) If reads are enabled and the file M86KRSVD.RWD is present in the current directory, that file is loaded.
- 5) Directories specified by the environment variable PATH are accessed in succession and the first readable M86KRSVD.RWD file encountered is read.

If the reserved word file is not found after doing all of the above checks in succession, an error occurs and M86K operation stops. Ordinarily, reserved word files are stored in the same directory as M86K.EXE. This file's contents are vital for normal M86K operation, and Sega will not accept any responsibility for problems with M86K operation if they are deleted or altered. For this reason, it is strongly recommended that this file be write-protected.



# **Errors**

---

Errors detected by M86K fall into three categories: fatal errors, (non-fatal) errors, and warnings. When a fatal error is detected, M86K immediately abandons execution. This level includes problems such as the working buffer being exhausted. When a non-fatal error is detected, the assembler continues to the end of the current pass (pass 1 or pass 2), then aborts. This level includes syntax errors in the source program. Warnings are issued for problems that do not warrant "error" classification, such as operands with values outside the permitted range; in this case M86K continues execution.

When a fatal error is detected, all output files which M86K is creating are not produced. When a non-fatal error occurs in pass 1, the files are not produced, but if a non-fatal error occurs in pass 2 the listing file only is output (when specified). The following is the format for showing errors.

```
filename(linenum): source line  
error message
```

Example

```
sample.asm(54): LD xyz  
xyz: undefine symbol
```

The symbol xyz is undefined.

## Warnings

The following are the warning messages which can be detected by M86K. In the list below, question marks indicate portions which may vary.

**???: bit number exceeds limits**

In a bit manipulation instruction, the number of bits exceeds the permitted range.

**absolute expression expected**

The expression must be able to be evaluated at assemble time.

**address beyond zero**

The value specified for the operand of an ORG instruction is negative.

**address exceeds limits**

The value specified for the operand of an ORG instruction exceeds the specified ROM capacity.

**address exceeds ROM size**

An address in assembled instruction exceeds the ROM capacity.

**chip name is different from one specified by CHIPNAME (???)**

The operand of a CHIP instruction is different from the current environment variable setting.

**END in included file**

An END pseudoinstruction was encountered in a source file specified by an INCLUDE pseudoinstruction.

**ENDF without FUNCTION**

ENDF was encountered while not in a function definition.

**ENDM without MACRO**

ENDM was encountered while not in a macro definition.

**EXITM outside MACRO**

EXITM was encountered while not in a macro definition.

**function code buffer overflow**

A function definition is too large, and will not fit in the buffer.

**illegal combination of attributes: ???**

The two sides of an arithmetic operator have incompatible attributes (bank or segment values).

**illegal style expression**

The operand expression of a SET or EQU is illegal.

**JMP/CALL placed at the end of memory block (FREE)**

A JMP or CALL instruction was encountered at an address with the bottom 12 bits equal to 0FFEh or 0FFFh. Since the segment alignment mode is "FREE" there may be no problem with the result of linking, but when the segment is aligned from the beginning of a memory boundary, the linker will give errors.

**Jump address is out of range (FREE)**

The destination address of a jump is outside a memory boundary. Since the segment alignment mode is "FREE" there may be no problem with the result of linking, but when the segment is aligned from the beginning of a memory boundary, the linker will give errors.

**LOCAL outside MACRO**

LOCAL was encountered while not in a macro definition.

**macro name in expression**

A symbol defined as a macro was encountered in an expression.

**macro name required**

The macro name is missing from a macro definition.

**no character in string**

Character was not found in a string constant.

**page width must be 72 ~ 132: ???**

The operand of a WIDTH instruction must be in the range 72 to 132.

**public ??? not defined**

A symbol declared as public is not defined.

**SET conflicts with PUBLIC**

An attempt has been made to reset a symbol declared as public with a SET.

### **symbol name required**

There is no symbol as the operand of a PUBLIC, EXTERN, or OTHER\_SIDE\_SYMBOL declaration.

### **undefined symbol in expression**

An undefined symbol occurs in an expression (only detected on pass 2).

### **value is out of range**

A value is outside the permitted range (the range depends on the operand).

### **zero divide: ??? modulo 0**

The right-hand operand of a MOD operator is zero.

### **zero divide: ??? / 0**

The right-hand operand of a division operator is zero.

## Non-Fatal Errors

The following are the non-fatal error messages which can be detected by M86K. In the list below, question marks indicate portions which may vary.

**???: 2, 8, 10 or 16 required**

The operand of a RADIX pseudoinstruction must be 2, 8, 10, or 16.

**???: constant required**

A numeric constant is missing.

**???: duplicated label**

A duplicate label occurred.

**???: duplicated symbol**

A duplicate symbol occurred.

**???: illegal character in numeric constant**

A numeric constant includes an illegal character.

**???: no such chip in the table**

A symbol specified in a CHIP instruction cannot be found in the reserved word file.

**???: open error**

An error occurred when opening a file.

**???: undefined symbol**

An undefined symbol has been referenced.

**???: radix violation**

A character in a numeric constant is improper for the base being used.

**???H, ???: out of internal RAM area**

The address allocation in a data segment is outside the permitted range.

**' not seen**

The closing single quote of a character constant is missing.

### **':' not seen**

In the explicit segment format of an EXTERN operand, the colon separating the segment and symbol is missing.

### **0x???: RAM address exceeds limits**

The address allocation of a data segment is outside the permitted range.

### **address duplicated**

There is a RAM address conflict specified by a DS pseudoinstruction.

### **address exceeds absolute limits**

The address of an assembled instruction exceeds 65535.

### **bank number should be 0~15**

The bank number must be in the range 0 to 15.

### **Branch address beyond zero**

The destination address of a branch is below zero (the start address of the current code segment).

### **Branch address exceeds limits**

The destination address of a branch exceeds the ROM capacity.

### **CSEG conflicts with WORLD EXTERNAL\_DATA**

It is not possible to have WORLD EXTERNAL\_DATA and a pseudoinstruction specifying a segment in the same source file.

### **CSEG isn't allowed in macro**

It is not possible to have a pseudoinstruction specifying a segment in a macro definition.

### **DS must be in DSEG**

A DS pseudoinstruction can only appear in a data segment.

### **DSEG conflicts with WORLD EXTERNAL\_DATA**

It is not possible to have WORLD EXTERNAL\_DATA and a pseudoinstruction specifying a segment in the same source file.

### **DSEG isn't allowed in macro**

It is not possible to have a pseudoinstruction specifying a segment in a macro definition.



**ELSE without IFxxx**

The conditional assembly IFxxx pseudoinstruction corresponding to an ELSE is missing.

**ENDF not seen**

The ENDF pseudoinstruction ending a function definition is missing.

**ENDIF without IFxxx**

The conditional assembly IFxxx pseudoinstruction corresponding to an ENDIF is missing.

**ENDM not seen**

The ENDM pseudoinstruction ending a macro definition is missing.

**external symbol can't be public**

An external symbol has been declared as public\_B

**Hardware configuration violation**

An instruction (such as CHANGE) corresponds to a function not implemented on the specified chip.

**identifier expected**

Something other than an identifier was encountered in a macro definition formal parameter list or EXTERN operand.

**illegal character in ??? constant**

A numerical constant includes a character illegal for the specified base.

**illegal character in binary constant**

A numerical constant includes a character illegal for the specified base.

**illegal symbol type**

A symbol type is improper for a PUBLIC declaration.

**illegal word in external list**

A syntax error occurred in an EXTERN operand.

**instructions can't be in DSEG**

Instructions other than DS cannot be included in a data segment.

### **JMP/CALL placed at the end of memory block (INBLOCK)**

A JMP or CALL instruction was encountered at an address with the bottom 12 bits equal to 0FFEh or 0FFFh. Since the segment alignment mode is "INBLOCK" the linker will give errors.

### **Jump address beyond zero**

The destination address of a jump is below zero (the start address of the current code segment).

### **Jump address exceeds limits**

The destination address of a jump exceeds the ROM capacity.

### **Jump address is out of range (INBLOCK)**

The destination address of a jump is outside a memory boundary. Since the segment alignment mode is "INBLOCK", the linker will give errors.

### **local symbol can't be public**

A local symbol has been given a public declaration.

### **lost SET symbol**

a symbol defined in a SET pseudoinstruction could not be found in pass 2. This may be an assembler internal error.

### **macro can't be public**

A macro has been given a public declaration.

### **maximum nesting of macro is 10**

Macros cannot be nested more than 10 deep.

### **Multiple WORLD specified**

More than one WORLD pseudoinstruction appears in the same source file.

### **name required for macro**

The name is missing from a macro definition.

### **no room for source line attribute object**

There is insufficient memory to store source line information (for debugging).

### **no value for EXT**

Even though a CHANGE instruction is used, a register EXT for an SFR is missing.

**not the symbol defined by SET**

An attempt was made to reset a value using SET, even though the symbol was not defined with SET.

**operand exceeds limits**

The repeat count in an REPT macro pseudoinstruction is not in the range 1 to 65535.

**ORG isn't allowed in macro**

The ORG pseudoinstruction may not appear in a macro.

**other-side symbol isn't allowed**

A symbol declared with OTHER\_SIDE\_SYMBOL cannot be specified here.

**other-side symbol isn't allowed here**

A symbol declared with OTHER\_SIDE\_SYMBOL cannot be specified here.

**other-side symbol or absolute constant is required**

A symbol declared with OTHER\_SIDE\_SYMBOL or constant is required.

**positive value required**

A negative value cannot be used.

**public ??? not defined**

A symbol declared as PUBLIC is not defined. This error is at the warning level when a PUBLIC symbol is neither given a value nor referenced, and at the (non-fatal) error level when the symbol is not given a value but is referenced.

**string is too long**

The length of a character string constant exceeds the limit (255 characters).

**symbol name required**

There is no symbol on the left-hand side of a SET or EQU.

**symbol not defined**

In pass 2, a symbol specified as a PUBLIC, EXTERN, or OTHER\_SIDE\_SYMBOL operand is missing. This may be an assembler internal error.

**syntax error**

A syntax error was detected.

### **syntax error near ???**

A syntax error was detected near ???.

### **too complexed expression for an operand**

An expression for an operand is too complex to be parsed.

### **too many CHIP pseudo operation**

There is more than one CHIP pseudoinstruction in a single source file.

### **too nested if-statements**

The nesting level of conditional assembly pseudoinstructions exceeds 10.

### **unbalanced conditional assembling controllers**

The source file ended while skipping in a conditional assembly.

### **unbalanced IF statement**

The source file ended while skipping in a conditional assembly.

### **unexpected end of file in string**

The source file ended while reading a character string constant.

### **unexpected end of line in string**

A line end was encountered while reading a character string constant.

### **unexpected EOF in conditional assembling**

The source file ended while skipping in a conditional assembly.

### **unexpected terminator ??? in conditional assembling**

The parser produced a fault while skipping in a conditional assembly. This may be an assembler internal error.

### **unmatched ELSE in skipping**

The source file ended while skipping in a conditional assembly.

### **unmatched ENDIF**

The source file ended while skipping in a conditional assembly.

### **WORLD conflicts xSEG**

It is not possible to have WORLD EXTERNAL\_DATA and a pseudoinstruction specifying a segment in the same source file.

## Fatal Errors

The following are the fatal error messages which can be detected by M86K. In the list below, question marks indicate portions which may vary.

**???(???): chip name not seen**

**???(???): chip name not seen.**

**???(???): decimal value required**

**???(???): hex-value and reserved-word are required**

**???(???): no chip name list**

**???(???): no reserved word seen**

**???(???): ROM size not seen**

**???(???): too many chip names**

**???(???): ???: unknown chip name**

**???(???): ???: unknown flag**

There is a syntax error in the reserved word file.

**???: illegal file name**

The specified file name contains an illegal character.

**???: no such chip in the table**

The chip name specified by the environment variable CHIPNAME was not found in the reserved word file.

**???: no such user**

The user name specified by ~user is missing.

**???: open error**

An attempt to open the specified file failed.

**???: unknown flag**

An unknown assembler option was specified.

### **???: unreadable**

The specified file cannot be read.

### **EMM v3.2 or later is required (v???.??? found)**

The EMS driver version is too old to support the application. Version 3.2 or later is required.

### **EMS allocation (??? pages) was failed**

An attempt to allocate EMS memory failed.

### **EMS deallocation was failed**

An error was detected when deallocating EMS memory.

### **flushing error in workfile**

An error was detected when flushing a workfile (no disk space left, or similar errors).

### **Getting EMM version was failed**

### **Getting EMS status was failed**

### **Getting free page count on EMS is failed**

### **Getting physical page frame address was failed**

An error was detected during the initialization of EMS memory, such as while checking the EMS driver version.

### **making temp. name for ??? failed**

An error was detected while giving the output file a temporary name.

### **Neither CHIP pseudo operation nor CHIPNAME environment variable were defined. Further execution aborted.**

Neither the CHIP pseudoinstruction nor the environment variable CHIPNAME specify a chip, so assemble cannot continue.

### **no more MAIN memory (???) ???**

Although there is a region which must be allocated to main memory, there is no more main memory left.

### **no more memory (???)**

There is no dynamically allocatable memory (main memory, EMS memory, or work file).

**no more NODE buffer (???) ???**

The working space used for parsing expressions is used up.

**no more PARAMETER buffer (???) ???**

The working space used for macro definitions and calling argument list processing is used up.

**no reserved word file available.**

An attempt to read in the reserved word file failed.

**no room for file: ???**

A file cannot be written because the disk is full.

**Pxxxx must be less than 65536**

The parameter buffer size must be specified to be 65535 or less.

**read error in workfile (???)**

An error occurred when reading a workfile.

**removing ??? failed**

After an error was detected, an attempt to delete a partially created file failed.

**renaming ??? ==\_ ??? failed**

An error occurred at the point of attempting to rename a file created with a temporary name with its real name. This can occur if a file with the new name already exists, and is read-only.

**too many file names**

Five or more file names are specified on the command.

**too many nested include files**

The nesting level of include files exceeds 10.

**unlinking work file is failed**

An error was detected attempting to delete a workfile.

**workfile ??? : already exist****workfile ??? : open error**

An error was detected creating a new workfile.

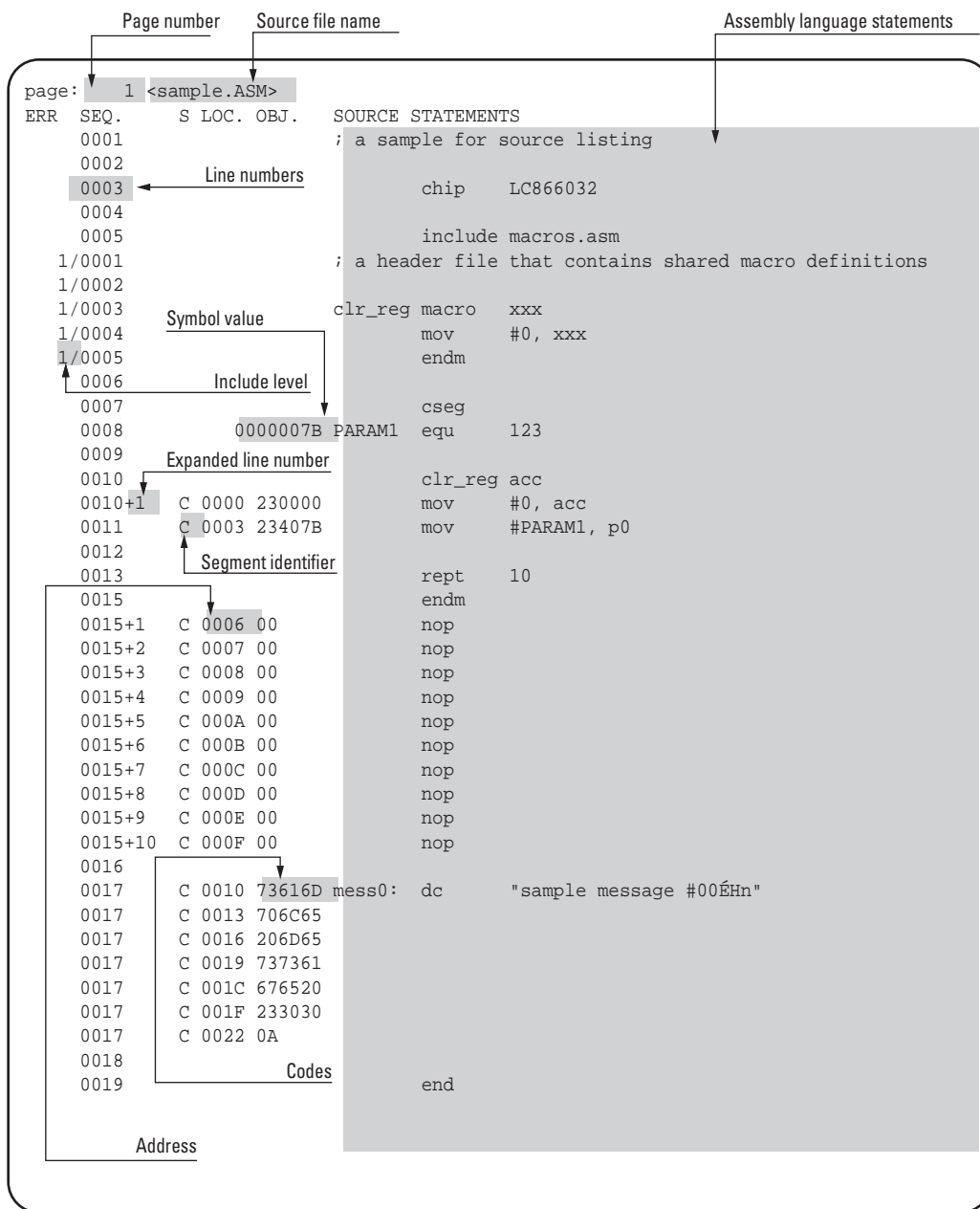




# ***Listing Format***

---

The format of the listing file created by M86K is shown below. This basically reproduces the content of the source file, with line numbers and machine code on the left. The arrangement of rows and columns is also designed to fit the page size. That is, a page is 60 lines (including header), and each line consists of 132 character positions (source lines exceeding this limit are folded), and the left part has fixed character columns. Tab characters in the source are converted to spaces, to preserve layout.



**Header**

This appears at the top of each page, and includes a blank margin, the page number, source file name, and headings for the columns of the listing.

**Page number**

The pages are numbered from 1.

**Source file name**

This is the name of the source file specified for assembly. If the specification includes a drive name or path, this is also shown.

**Assembly language statements**

This shows the text of the source file, macro expansions (when their output is not suppressed), and include files.

**Line numbers**

These are line numbers (in decimal) in the source file. When the code section occupies more than one line in the listing the source file line numbers are repeated.

**Include level**

This shows the nesting level of include files. Lines from the source file itself have no include depth shown. Then the contents of an included file are shown as depth 1, and nested includes as 2, and so forth. The slash separates the include level from the line number.

**Symbol value**

When a symbol is given a value, if that value is explicit at assemble time it is shown here, as an 8-digit hexadecimal value. If the value is not known, it is not shown.

**Expanded line number**

This indicates a line not present in the source file, which has been produced by a macro expansion. The extra lines are numbered from 1 for each macro expansion.

**Segment identifier**

This character indicates whether the corresponding line is assembled as CSEG or DSEG code. Uppercase 'C' indicates CSEG INBLOCK, lowercase 'c' indicates CSEG FREE, and D indicates DSEG.

**Address**

When a line generates CSEG or DSEG code, this shows the address of the first byte as a four-digit hexadecimal value. This address is an offset from the beginning of the segment.

**Codes**

When code to be written to ROM is generated by assembling a source line, it is shown here, as two hexadecimal digits for each byte. A maximum of three bytes are shown on each line, in order of increasing address. When more than three bytes are assembled, they are shown on subsequent lines with the same source line number.



# ***Specifying Files for Linking***

---

There are two ways of specifying the parameters when starting L86K.

- 1) Specifying all parameters on the L86K command line
- 2) Specifying parameters by responding to the prompts displayed by L86K

To force an exit from L86K, press the following key combinations.

<b>Computer type</b>	<b>Keys</b>
PC/AT compatible	Ctrl + C or Ctrl + Pause/Break
NEC PC-9800 series	CTRL + C or Stop

## Specifying File Names

When specifying file names to L86K, either in the command line, or in response to the prompts, case is ignored. For example, the following three file names all refer to the same file.

```
sample.obj  
SAmpLE.OBJ  
SAMPLE.OBJ
```

If a file name is specified without an extension, L86K supplies the following default extensions.

<b>File format</b>	<b>Default extension</b>
Object file	.OBJ
Executable file	.EVA
Library file	.LIB
Option file	.OPT
Font file	.CGR
Flash memory data file	.Hnn

The number nn is that specified by the -B option.

## Specifying Parameters on the Command Line

```
L86K_options_objectfiles_,_evafile_,_libraryfile____;_
```

### options

Specify linkage loader options as listed in Chapter, “Option Switches.” If options are specified they must come before other parameters.

### objectfiles

Specify the object files to be linked, the linking start address, and library file names. At least one file name is required. When specifying multiple files, separate them by space characters. If the file names do not fit into one line, add a plus sign (+) to the end of the line to indicate a continuation. If the object file extension is omitted, the default extension .OBJ is added. If a file extension is specified, it takes precedence. In either case, a drive name and full path specification is possible. If .LIB is specified as the file name extension, the library is linked as well.

### evafile

Specify a file name for execution on (downloading to) EVA86K. If no name is specified, the first file name specified in objectfiles is used, with the file name extension changed to .EVA.

---

**Caution:** EVA86K is a device for emulating the LC86K series on the computer. For Visual Memory, the EVA86K data is converted to HEX format, and run on the Visual Memory simulator.

---

### libraryfile

Specify a library name. If no library is required, the specification can be omitted.

---

**Caution:** If a DUMMY.OBJ file is attached to the Visual Memory SDK, link this file in as well.

---

### Example

```
C> L86K MAIN SUB0 SUB1,TEST,TEST.LIB ø
```

This links the object modules MAIN.OBJ, SUB0.OBJ, SUB1.OBJ, MAIN.OPT, and MAIN.CGR. If there are any symbols remaining undefined after linking MAIN.OBJ, SUB0.OBJ, and SUB1.OBJ, the linker searches TEST.LIB for these symbols, and links the appropriate modules.

# Specifying Parameters at the Prompts

Enter the following command to start the linkage loader without specifying any file names.

```
L86K_moptions_n
```

L86K produces the following prompts, one line at a time. Enter the parameters as required.

```
SANYO (R) LC86K series Linkage Loader Version 4.00  
Copyright (c) SANYO Electric Co., Ltd. 1989-1995. All rights reserved.  
Object modules.OBJ]:  
EVA filenamebasefilename.EVA]:  
Libraries.LIB]:  
Option filenamebasefilename.OPT]:  
Font filenamebasefilename.CGR]:
```

L86K waits for a response before displaying the next prompt. Section 3.1, "Specifying File Name" describes the conventions for responding to the prompts.

The responses to the L86K command correspond to the parameters on the L86K command line, except for Option filename and Font filename. These are described below.

---

**Reference:** For details of the L86K command line, see Section, "Specifying Parameters on the Command Line."

---

Prompt	Command line argument
Object modules	objectfiles
EVA filename	evafilename
Libraries	libraryfiles

When using the L86K prompts, enter the above four items, then the following two items.

### Option filename

Enter the name of the option file for the chip for which the EVA file will be created.

### Font filename

Enter the font file name.

If the last character typed on the response line is a plus sign (+), the prompt returns another line, so the input can be continued. In this case the plus sign must come at the end of a complete file or library name, path name, or drive name.

---

**Caution:** This file specification is not necessary for Visual Memory. This file specification is not necessary for Visual Memory.

---



**Default Responses**

To select the default response to the current prompt, press the Enter key, without entering a file name. The next prompt appears.

To select the default response to the current prompt and all of the remaining prompts, enter a semicolon (;) and press the Enter key. This disables input to the rest of the prompts for this linking session, and saves time when using the default settings. Since, however, there is no default for the Object modules prompt, it is not possible to enter a semicolon for this prompt.

The following table shows the defaults for the L86K prompts.

<b>Prompt</b>	<b>Default</b>
EVA filename	The name of the first file in the response to "Object modules," with the file extension changed from .OBJ to .EVA.
Libraries	No libraries are searched.
Option filename	The file name specified in response to the "EVA filename" prompt, with the file extension changed from .EVA to .OPT.
Font filename	The file name specified in response to the "Option filename" prompt, with the file extension changed from .OPT to .CGR.

# Files Referenced During Linking

L86K always references the following files during linking.

---

**Caution:** For Visual Memory, the system BIOS used for reading and writing flash memory is in ROM, and therefore the following do not necessarily apply.

---

LC86K.LIB

System information is stored in LC86K.LIB. At linking time, L86K gets system information for the target CPU from LC86K.LIB, and stores it in the EVA file.

It also references the following option file during linking.

LCnn00.OPT ... nn is a two-digit value corresponding to the device type

LC86K.LIB and LIBnn00.OPT must be present in the same directory as L86K.EXE or in a directory to be found in the PATH environment variable.

# Option Switches

This chapter describes how to use the linkage loader options to control the operation of L86K. The options are all introduced with the linkage loader option character, which is slash or minus.

## **-B=bank number**

### **Creating a flash memory HEX file for the LC86800 series**

The -B option specifies the bank number for LC86800 series flash memory (WORLD\_EXTERNAL\_DATA). The bank number is written as a hexadecimal value from 1 to FF. Here the bank number is the data file extension.

---

**Caution:** Only bank 0 of flash memory is available to applications in Visual Memory.  
Do not specify this option.

---

## **Example**

```
C> L86K /B=1 SAMPLE;
```

In this case the data file created is SAMPLE.H01.

## **-C=address**

### **Specifying a CSEG loading address**

The -C option applies to the immediately following object module, and specifies a code segment loading address. The address is written in hexadecimal.

If this option is omitted, L86K may load the object module code segment at any address.

## **-D=address**

### **Specifying a DSEG loading address**

The -D option applies to the immediately following object module, and specifies a data segment loading address. The address is written in hexadecimal.

If this option is omitted, L86K may load the object module data segment at any address.

### **-E**

#### **Allowing DSEG address overlaps**

If the -E option, more than one symbol can be defined in data segments at the same address, without causing an error.

### **-I**

#### **Do not distinguish case**

By default, L86K treats lowercase and uppercase letters as distinct. If the -I option is specified, case is ignored.

### **-P**

#### **Create a loading map**

Specifying the -P option creates a file containing a listing of the linker mappings (the linking state of each segment, and public symbol locations). This map file has the file name specified on the command line or in the prompts for the EVA file, with the extension changed to .MAP. If, however, a fatal error prevents linking from continuing, the map file is not created.

---

**Note:** A map file will be necessary when using the reverse assembly function of the Visual Memory simulator to display address labels.

---

### **-L**

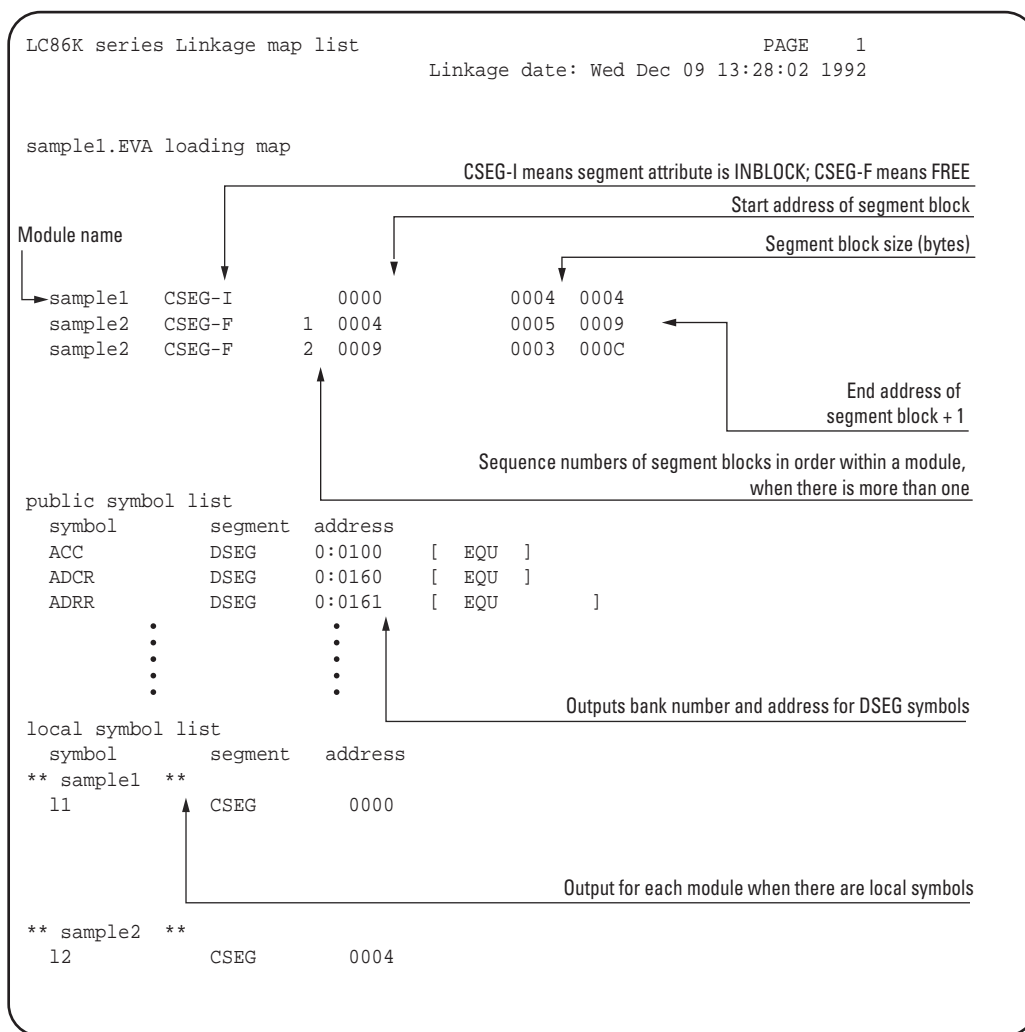
#### **Create a local symbol listing**

The -L option is only valid in combination with the -P option, and adds a listing of local symbols for each module in the map file.

---

**Note:** A map file will be necessary when using the reverse assembly function of the Visual Memory simulator to display address labels.

---



**-W**

**Issue warning messages for operand values**

When the -W option is specified, operands of JMP instructions and so forth are checked to be in range (for a JMP, the low 12 bits of the operand are allowed, and the value must thus be in the range 0 to 4095). If not, a warning message is issued (only for the number of valid bits stored in the instruction code; overflow bits are discarded).

**-A, -F, -O, -R**

**Optimizing loading of CSEG FREE blocks**

Normally L86K links the code segments in the order in the object modules and following the order of specification on the command line, and at this point L86K aligns the executable file segment data (code segments) on 4096-byte boundaries. For this reason, gap may appear between code segments.

To minimize this wasted space, there are four types of optimization of segment block positioning to use memory more efficiently.

The following are the loading methods.

### **-A option**

With the -A option all code segment blocks to be linked are loaded in decreasing order of size. If a segment block that crosses a 4096-byte boundary has the INBLOCK attribute, it is aligned on a boundary; if it has the FREE attribute it is not repositioned.

### **-F option**

The -F option is only valid in combination with -A. After code segment blocks with the INBLOCK attribute are linked in the order specified on the command line, code segment blocks with the FREE attribute are located in gaps occurring as described above (if there are no regions to locate code segment blocks with the FREE attribute, they are located from the last address, in decreasing order of size).

### **-O option**

The -O option is only valid in combination with -A. After first linking code segment blocks with the INBLOCK attribute in decreasing order of size, code segment blocks with the FREE attribute are located in empty regions (if there are no regions to locate code segment blocks with the FREE attribute, they are located from the last address, in decreasing order of size).

### **-R option**

The -R option is only valid in combination with -A. After first linking code segment blocks with the INBLOCK attribute in decreasing order of size, code segment blocks with the FREE attribute are located in empty regions. At this stage, if two consecutive 4096-byte regions include empty space, the later code segment block with the INBLOCK attribute is repositioned at the end of the region, and a code segment with the FREE attribute is located in the empty region (if there are no regions to locate code segment blocks with the FREE attribute, they are located from the last address, in decreasing order of size).

### **-S**

#### **Specify symbol sorting**

If the number of public symbol definitions in the linked object files (including SFR definitions in the file LC86K.LIB) exceeds 8192, or the number of local symbol definitions in the linked object files exceeds 8192, then because of the drop in processing speed for symbol sorting, the following messages appear, showing the progress of the sorting.

```
Public(Local) symbol table: Sorting. nn / nn blocks  
Public(Local) symbol table: Sorting(merge).. nn %
```

However, if the -S option is specified, in this case symbol sorting is not carried out and linking is abandoned, with the following error message.

# ***Object Alignment***

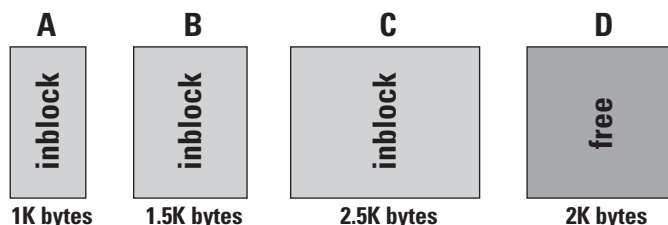
---

As described under “Optimizing loading of CSEG FREE blocks” in Chapter 9, “Option Switches” (the -A, -F, -O, and -R options), when optimization is specified for linking, objects are aligned differently from the normal case. The are four types of optimization, and the corresponding object alignment for each of the types is described below.

## -A option

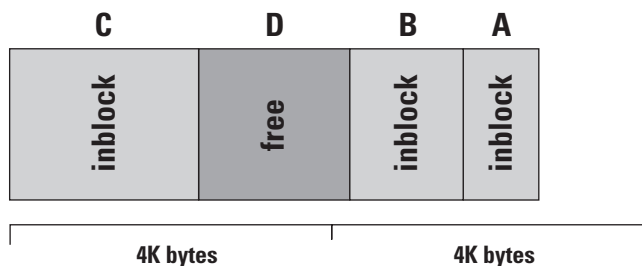
There are two alignment specifications for code segments: CSEG INBLOCK (aligned within a 4096-byte block) and CSEG FREE (align regardless of 4096-byte boundaries). When the -A option is specified segments are placed in the best positions in order of decreasing size, taking the INBLOCK and FREE segments together (the INBLOCK segment are located within 4096-byte blocks, but the FREE segments are aligned freely).

Consider, for example, the following objects:

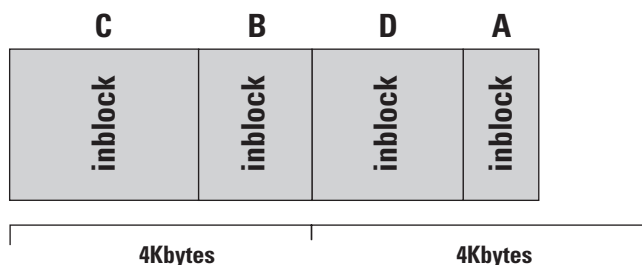


If the -A option is specified for linking, then the segments with INBLOCK/FREE specifications are located in decreasing order of size as follows.

```
L86K -A A B C D;
```



In this example, the segment straddling a 4096-byte boundary is object D, which has the FREE attribute, thus not requiring alignment. If object D has the INBLOCK attribute, the following is the result.

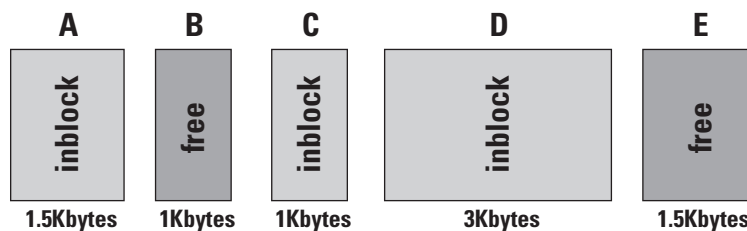




## **-A -F options**

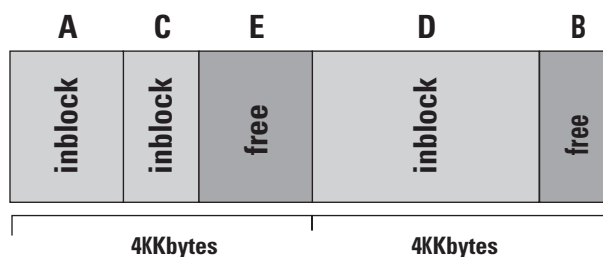
When **-A-F** is specified, the segments with the **INBLOCK** attribute are located in their order on the command line, then segments with the **FREE** attribute are located in decreasing order of size, in the best available positions.

Consider the following set of objects.



When these are linked with **-A-F** specified, A, C, and D are located in the command line order (with D aligned to a 4096-byte boundary), then E and B in sequence in the best available positions.

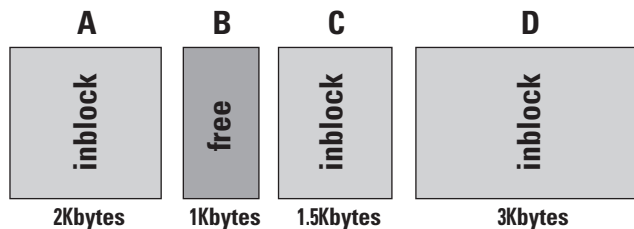
```
L86K -A-F A B C D E;
```



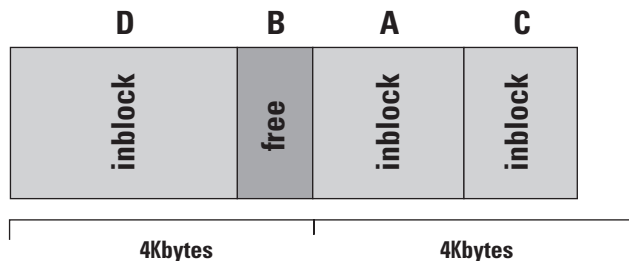
## -A -O options

When -A-O is specified, the segments with the INBLOCK attribute are located in decreasing order of size, then segments with the FREE attribute are located in decreasing order of size, in the best available positions.

Consider the following set of objects.



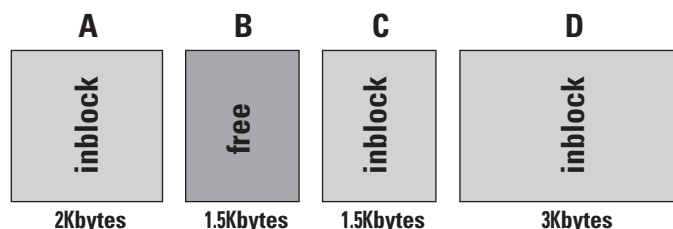
When these are linked with -A-O specified, D, A, and C are located in the best available positions in order of decreasing size, then B is located in the best available position.



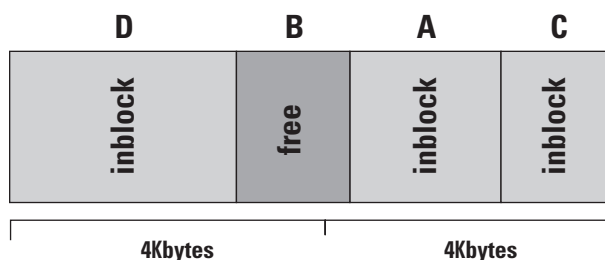
## **-A -R options**

When **-A-R** is specified, the segments with the **INBLOCK** attribute are located in decreasing order of size, then if two consecutive 4096-byte regions include empty space, the segments in the second of the two regions are realigned to bring the empty space together, and segments with the **FREE** attribute are inserted in this space. alignment the later code segment block with the **INBLOCK** attribute is repositioned at the end of the region, and a code segment with the **FREE** attribute is located in the empty region (if there are no regions to locate code segment blocks with the **FREE** attribute, they are located from the last address, in decreasing order of size).

Consider the following set of objects.



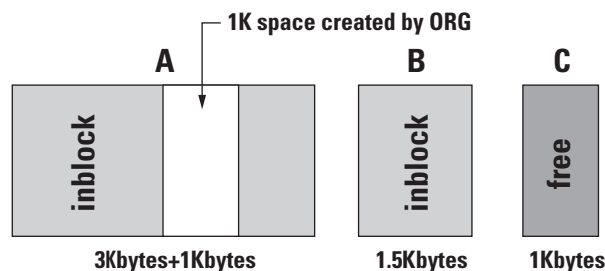
When these are linked with **-A-R** specified, after locating **D**, **A**, and **C**, segments **A** and **C** are realigned with the end boundary of their 4096-byte block, to open up a space, in which **B** can be located.



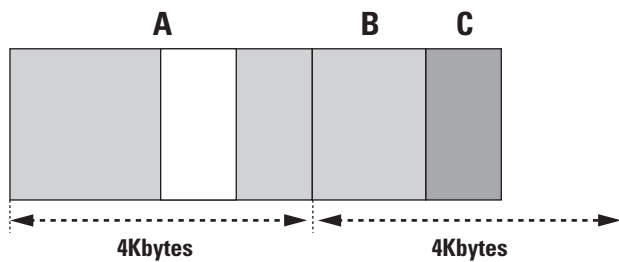
In all optimization operations, if an object module includes a space created by use of an **ORG** pseudoinstruction, this space is made available for optimized segment alignment.

The following is an example of this.

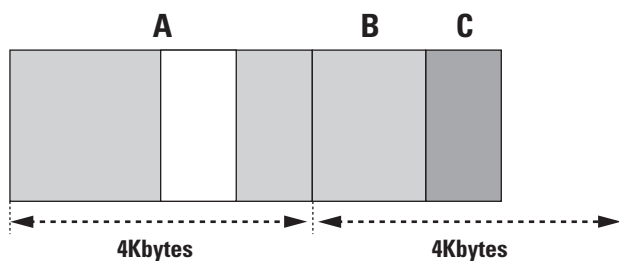
Consider the following set of objects.



If these are linked without optimization the following is the result.



When these are linked with optimization, the space created within segment A by the ORG pseudoinstruction is filled as follows.



If a loading address (-C option) is specified together with the optimization specification, then if the first segment of the file has a FREE attribute, the loading address specification is followed for this segment only (subsequent FREE blocks are optimized).

# ***Errors***

---

## **Fatal Errors**

When a fatal error is detected during linking, L86K displays a message, and aborts processing. The following are the L86K error messages.

### **Chip name unmatched**

An attempt was made to link object modules for different chips.

### **Data file specified**

A data file was specified for EVA file creation.

### **Data segment size exceeds**

An attempt was made to link a DSEG object exceeding the RAM size.

### **External undefine symbol**

An external symbol is undefined.

### **Illegal bank number specified**

The specified flash memory bank number is incorrect.

### **Illegal file format**

The specified file is not for the LC86K series.

### **Illegal option specified**

An illegal option was specified.

**Internal module not specified**

In linking an internal program file was not specified.

**Loading address multiple assignment**

More than one object has been allocated to the same address.

**No such file or directory**

The specified file does not exist.

**Program file specified**

A program file was specified when creating a flash memory data file (bank 1).

**Public symbol multiple define**

A public symbol is multiply defined.\_

**Segment size exceeds**

An attempt was made to link when exceeding the segment size.

**WORLD attribute unmatched**

A program file with the WORLD INTERNAL or WORLD EXTERNAL attribute and a data file with the WORLD EXTERNAL\_DATA attribute are combined.

## Non-Fatal Errors

If a non-fatal error is detected during linking, a message appears, but linking continues. L86K produces the following warning messages.

**Cannot access file: LC86K.LIB**

The library file LC86K.LIB containing the reserved words does not exist. LC86K.LIB contains the reserved words for each chip, and must be present in the current directory or a directory specified in the PATH environment variable.

**Module not in library**

A reserved word for the target chip is not present in the file LC86K.LIB.

**Operand data overflow**

The value specified in the operand field exceeds the designated range. (Range varies according to statement.)

**Operand data type mismatch**

An illegal segment symbol was specified in the operand field.

# Starting the Program

There are two ways of starting LIB86K and specifying the parameters.

- 1) Specifying all parameters on the LIB86K command line
- 2) Specifying parameters by responding to the prompts displayed by LIB86K

To force an exit from M86K, press the following key combinations.

Computer type	Keys
PC/AT compatible	Ctrl + C or Ctrl + Pause/Break
NEC PC-9800 series	CTRL + C or Stop

## Specifying File Names

When specifying file names to LIB86K, either in the command line, or in response to the prompts, case is ignored. For example, the following file names all refer to the same file.

```
sample.obj
SAmplE.OBJ
SAMPLE.OBJ
```

If a file name is specified without an extension, LIB86K supplies the following default extensions.

File format	Default extension
Library file	.LIB
Object file	.OBJ
Listing file	None

## Specifying Parameters on the Command Line

```
LIB86K_option_ oldlibrary_commands __, _listfile __, _newlibrary ____; _
```

### option

The only option which can be specified is /?.

### oldlibrary

Specify the library file to be processed. This parameter cannot be omitted. If the library file extension is .LIB, it can be omitted. However, if the user's library file extension is other than .LIB, it cannot be omitted. There is no default for the library file, so if this parameter is missing an error message results. If the specified library file does not exist, the following prompt appears.

```
Library file does not exist. Create? (y/n)
```

To create a new library, enter "Y". If any character other than "Y" is entered, the library manager terminates. Entering an existing library file with just a semicolon carries out a consistency check on the library. This checks whether all of the modules within the library can be used. If an error is found, an error message is produced.

### commands

The commands parameter includes symbols such as +, -, ++, \*, and -\*, which are used to control the program operation. One object file name or module name can be specified for each command, to carry out a number of operations. If the commands are omitted, no changes are made to the library file.

Command	Meaning
+	Add a module. The module in the object file specified following the command is added at the end of oldlibrary. This command cannot be used to merge libraries.
-	Delete a module. The module specified following the command is deleted from oldlibrary.
++	Replace a module. The module in the object file specified following the command is added at the end of oldlibrary, and the existing module of the same name is deleted.
*	Copy a module. A search is made in oldlibrary for the module specified following the command, and this is written to an object file of the same name. The copied module is left in oldlibrary.
-*	Move a module. A search is made in oldlibrary for the module specified following the command, and this is written to an object file of the same name, but the module is deleted from oldlibrary.

### listfile

In listfile, specify a file for output of a list of the public symbols, external reference symbols, and module names in the library. If omitted the listing is sent to standard output.

### newlibrary

The newlibrary parameter specifies an output library name. If this is omitted, the existing version of the oldlibrary file has the extension changed to .BAK, and the new library is written to the file oldlibrary.



## Option

Specify the `/?` option to display a help message.

## Examples of Command Line Execution

### Example 1

```
LIB86K HOME-+ROM;↵
```

This example deletes the module ROM from the library HOME, and adds the object file ROM.OBJ at the end of the library.

### Example 2

```
LIB86K HOME-ROM+ROM;↵  
LIB86K HOME+ROM-ROM;↵
```

In the first version, the module ROM is deleted from the library HOME, and then the object file ROM.OBJ is added at the end of the library. In the second version, the ROM.OBJ object file is added to the library HOME first, and then the ROM module is deleted. Therefore, in the first version, the ROM module remains in the library, but in the second version it does not. This is because the command symbols are executed in the order they are specified.

### Example 3

```
LIB86K HOME,LCROSS.PUB ↵
```

After carrying out a consistency check on HOME.LIB, a cross-reference listing is written to the file LCROSS.PUB.

### Example 4

```
LIB86K FIRST -*STUFF*MORE,,SECOND ↵
```

The module STUFF is extracted from the library FIRST.LIB and written to the file STUFF.OBJ, and then the module STUFF is deleted from the library. The module MORE is written from the library to the file MORE.OBJ, but remains in the library. The rewritten library is named SECOND.LIB, and corresponds to FIRST.LIB, with the STUFF module deleted.

# Operation with the Prompts

Enter the following command to start LIB86K without specifying any parameters. Then follow the prompts from the assembler to enter the parameters.

```
LIB86K_moption_n↵
```

LIB86K displays the following prompts, one at a time.

```
Library name:
Operations:
List file:
Output library:
```

After displaying each prompt, LIB86K waits for user input. After user input, it displays the next prompt, and waits again.

The responses to the prompts correspond to the parameters entered on the command line, as shown in the following table.

Prompt	Corresponding command line parameter
Library name	Corresponds to the oldlibrary parameter. If the library name is followed by a semicolon, LIB86K carries out a consistency check.
Operations	Corresponds to the commands parameter.
List file	Corresponds to the listfile parameter.
Output library	Corresponds to the newlibrary parameter.

## Prompt Line Extension

In response to the Operations prompt, entering an ampersand (&) at the end of the line produces another Operations prompt, so that the specification can be continued.

## Default Responses

Except for the Library name prompt, there are default values, which are used when the response to the prompt is a semicolon or the Enter key. The following table shows the prompt default values.

Prompt	Default value
Operations	Make no changes to the library file.
List file	Select standard output for the listing. No list file is created.
Output library	The output library name is the same as the original library name.

# ***Error Messages***

---

This chapter lists error message and their meanings.

**cannot access file**

LIB86K cannot open the specified file.

**cannot create new library**

The disk or root directory is full, or the library file already exists and is read-only.

**cannot rename old library**

The .BAK version is read-only, and LIB86K cannot rename the old library with the .BAK extension.

**comma or newline missing**

A comma or newline is missing on the command line.

**error reading from library**

LIB86K cannot read data from the specified library file.

**error writing to new library**

The disk or root directory is full

**insufficient memory**

There is insufficient memory for LIB86K to run.

### **interrupted by user**

LIB86K execution was interrupted by the user.

### **invalid library header**

The input library file is in an invalid format.

### **module not in library\_ ignored**

The specified module to be replaced was not found in the library.

### **output-library specification ignored**

In the case of a new library name, an output library was also specified.

### **syntax error : illegal file specification**

A command operator such as a minus sign (-) is not followed by a module name.

# Cross-Reference

---

This chapter describes the cross-reference listing format.

LC86K series Library Analysis List

PAGE 1

Tue Feb 18 13:56:12 1992

Number of Module count: 2      Library create date: Wed Oct 16 15:34:53 1991  
 Library update date: Tue Feb 18 10:55:23 1992

Including Modules: 1      2

Module name: 1      Source name: 1.ASM  
 Assembler name: SASM 1.0      Assembly date: Tue Oct 22 15:54:43 1991  
 Target chip name: LC868700  
 Including Public symbols:  
 Including External symbols:  
 Test      sample      label1

Module name: 2      Source name: 2.ASM  
 Assembler name: SASM 1.0      Assembly date: Tue Oct 22 15:54:43 1991  
 Target chip name: LC868700  
 Including Public symbols:  
 Including External symbols:  
 label1      label2      label3



# ***Starting the Program***

---

Using E2H86K to convert an EVA format file into a HEX format file produces files with the extensions .HEX and .H00.

The file with the extension .HEX is a 64K-byte flash memory file in HEX format. The region not used for the program is filled with zeros. This file is not normally required.

The file with the extension .H00 is a HEX format file including only the program itself. This file can be read into the Visual Memory simulator, or converted to a BIN format file with H2BIN.EXE, for reading into a Visual Memory device.

## Specifying File Names

When specifying file names to E2H86K, case is ignored. For example, the following three file names all refer to the same file.

```
sample.eva  
SAmpLE.EVA  
SAMPLE.EVA
```

If a file name is specified without an extension, E2H86K supplies the following default extensions.

File format	Default extension
EVA file	.EVA
HEX file	.HEX



## Specifying Parameters

E2H86K\_option\_ EVA\_filename \_HEX\_filename\_

### **option**

Specify the option as described in Section , "Option Specification,". This must immediately follow the command name.

### **EVA\_filename**

Specify the debugged file (with the extension .EVA). This is referred to as the EVA file.

### **HEX\_filename**

Specify the name of an Intellec HEX format file. If HEX\_filename is omitted, the same file name as the EVA\_filename is used. When a flash memory data file is converted the extension is .H00.

---

**Caution:** There is no prompt option.

---

### **Example 1**

```
C>E2H86K PROG012
EVA file ____HEX file, _flash memory HEX file
PROG012.EVA____PROG012.HEX, PROG012.H00
```

### **Example 2**

```
C>E2H86K ø
```

This displays the following help message.

```
SANYO LC86000 Series EVA-file to HEX-file generator V1.00A
Copyright (C) SANYO Electric Co.,Ltd. 1992
Usage: e2h86k options] EVA_filename HEX_filename]
Options: /I ... information on/off (default: on)
```

## Option Specification

The option must be introduced with a slash (/).

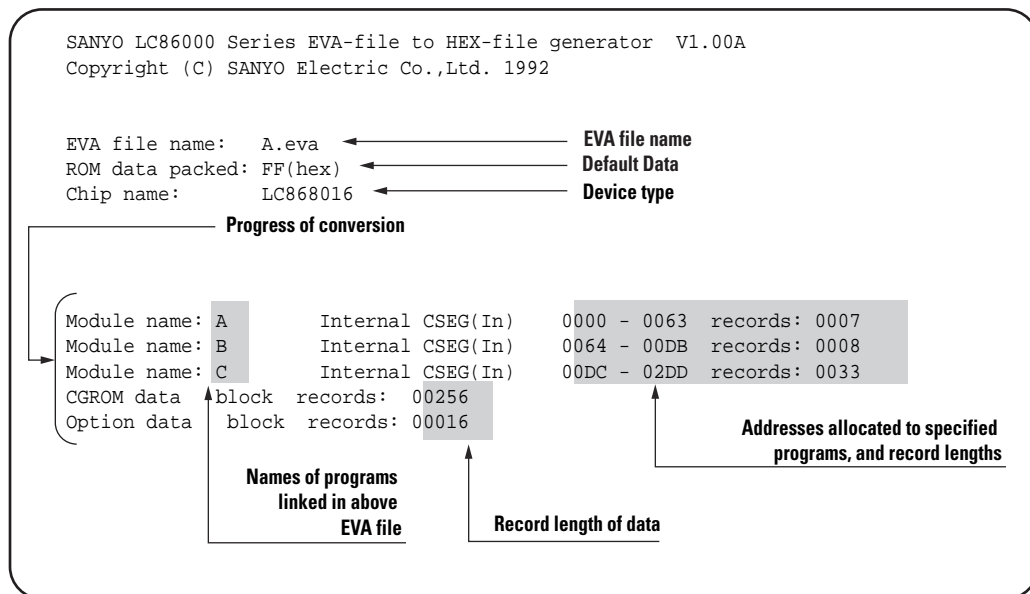
**Caution:** It is not possible to use a minus sign (-).

**/I**

### Suppress information display during conversion

The /I option suppresses the display of information during conversion of the EVA file to the HEX file. If this option is not specified, the progress of the conversion is indicated.

### Example display



# ***Error Messages***

---

## **Fatal Errors**

If E2H86K detects a fatal error during operation, it displays one of the messages listed below, and terminates.

Error message: Fatal error : ... message ...

**'filename' File not close.**

The file 'filename' cannot be closed.

**'filename' File not create.**

The file 'filename' cannot be created.

**'filename' File not open.**

The file 'filename' cannot be found.

**'filename' not EVA file format.**

The file 'filename' is not in EVA format.

**'filename' user disk full.**

The disk became full while writing the file 'filename.'

**Chipname undefined.**

The chip name in the EVA file is not known.

**ROM size over. (ROM size: XXXX)**

The program size exceeds the ROM size.

**Tablename allocation error.**

There is insufficient memory to reserve space for tablename.



# ***Starting the Program***

---

Using H2BIN, a BIN format file for reading into the Visual Memory device can be created. The BIN format file can be transferred to Visual Memory using a development computer, Dev. Box, and the Memory Card Utility special-purpose transfer utility.

Alternatively, using the Shinobi library backup functions, the BIN format file can be transferred to Visual Memory.

---

**Reference:** For the method of transfer using a development computer, Dev. Box, and the special-purpose transfer utility, see “Visual Memory Tutorial.”  
For the method of transfer using the Shinobi library, refer to the `buSaveExecFile()` function under “Backup Functions” in the SEGA library manual, Vol. 2.

---

## **Specifying File Names**

When specifying file names to H2BIN, case is ignored. For example, the following three file names all refer to the same file.

```
sample.H00  
SAmple.H00  
SAMPLE.H00
```

## Specifying Parameters

```
H2BIN _HEX_filename_ _BIN_filename_ ↵
```

---

**Note:** H2BIN has no command line options.

---

### HEX\_filename

Specify the file created with E2H86K with the extension .H00.  
Starting H2BIN without specifying a file name produces a brief help message.

---

**Note:** This file must be in the Intellec HEX format.

---

### BIN\_filename

Specify the name for the BIN format file. If BIN\_filename is omitted, the same file name as the HEX\_filename is used, with the extension .BIN.

---

**Caution:** H2BIN does not run in the full screen mode of MS-DOS. If started in the full screen mode, it forcibly switches to the window mode.

---

This is important when using MAKE or a batch operation.

### Example 1

```
C>H2BIN PROG012.H00 ↵
HEX file ____BINfile_
PROG012.H00__PROG012.BIN
```

### Example 2

```
C>H2BIN ↵
```

The following message (brief help) appears.

```
INTELLEC HEX to binary converter Version 0.10 SEGA SYSTEM R&D
Usage:          H2BIN <HEX file name > [<binary file name>]
                HEX file name is the source file
                Binary file name is the output file (can be omitted)
Function:       converts INTELLEC HEX format file to binary.
                If the output file name is omitted, it is the source file name with the
                extension changed to .BIN.
Example:        C:\VM_SDK\LC86K\H2BIN.EXE lcd_puu3.hex lcd_puu3.bin
```

# ***Error Messages***

---

## **Fatal Errors**

If H2BIN detects a fatal error during operation, it displays one of the messages listed below, and terminates.

**file 'filename' not found.**

The specified HEX format file cannot be found. Check the path and file name. Do not omit the extension .H00.

**file 'filename' cannot be created.**

A write to the specified BIN format file failed. Check the available disk space, and whether other applications have the file open.

**Conversion failed.**

The HEX format file CRC is bad. The HEX file may be corrupted. Run E2H86K.EXE to recreate the HEX format file.

**Not an INTELLEC HEX format file.**

The HEX format file is of a different type. Run E2H86K.EXE to recreate the HEX format file.

**error: extended address code detected - not currently supported.**

The HEX format file is of a different type. Run E2H86K.EXE to recreate the HEX format file.

**error: unknown record type detected.**

The HEX file may be corrupted. Run E2H86K.EXE to recreate the HEX format file.





# ***Overview of MAKE***

---

MAKE automates program development. It automatically updates an executable file (EVA) when a source file (ASM), object file (OBJ), option file (OPT), CGROM file (CGR), and so on is updated.

To run MAKE requires a file, the makefile, including the necessary information. This is a text file, of the instruction used to build the program. These instructions include generation rules, macros, directives, and implicit rules. A generation rule consists of a target, and files it depends on, together with commands for building the target. MAKE compares the time stamp of the target with the time stamp of the files on which it depends, and if any of these have been updated more recently than the target, uses the specified command to rebuild the target.

## Running MAKE

To start MAKE, enter the following command.

```
MAKE [options] [/f makefile] [/x errorfile] [targets]
```

### **options**

Enter any MAKE options. For details, see Section 19.1.2, “Command Line Options.”

### **makefile**

Specify the name of the makefile. Note that a space is required between '/f' and 'makefile'. The makefile name may be omitted if it is 'MAKEFILE.'

### **errorfile**

Specify a file for output of errors. Note that a space is required between '/x' and 'errorfile'. The error output is normally to the display, but with this option can be to a file.

### **targets**

Specify the target or targets to be built. If this specification is omitted, and there is no .TARGET directive, the first target in the makefile is built.

## Build Priority Sequence

MAKE looks for the rules for a build in the following priority sequence.

- 1) If the /f option is specified, MAKE looks for the specified makefile in the current directory or specified directory. If the file is not found, MAKE terminates.
- 2) If the /f option is not specified, MAKE assumes the file name is 'MAKEFILE,' and looks in the current directory.
- 3) Unless the /r option is specified, MAKE looks for a makerule file in the current directory. If not found in the current directory, MAKE looks in the directory containing MAKE itself. If the makerule file cannot be found, MAKE terminates.

## Command Line Options

The following options control the operation of MAKE. The option letters are not case-dependent, but are always preceded by a slash (/).

### **/E**

#### **Give priority to external macros**

When referencing a macro, give priority to an external macro. The default is to give priority to an internal macro.

**/I**

### **Ignore result codes, and continue processing**

This ignores the result code from a command specified in the makefile. MAKE continues processing to the end of the makefile, even if errors occur. To ignore result codes in a particular part of the makefile only, use the hyphen (-) command modifier or .IGNORE directive.

**/N**

### **Show sequence of build without executing commands**

This just displays the commands which would be executed in the makefile build, without actually carrying them out. This is useful for makefile debugging, and for checking which target files should be updated.

**/R**

### **Do not read rule file**

When this option is specified, MAKE will not read a makerule file. The default is to allow the makerule file to be read.

**/S**

### **Suppress command display**

Do not echo commands in the makefile. To suppress command display for parts of the makefile only, use the at sign (@) command modifier or .SILENT directive.

**/?**

### **Show help**

Display the MAKE command line syntax.

# Makefile Syntax

The makefile is a text file: create it using a text editor. Normally the makefile is called "MAKEFILE," but if there are a number of different makefiles, they can have distinct names. The makefile contains generation rules, macros, implicit rules, and directives.

## Generation Rules

The generation rules are the core of the makefile. They are written as follows.

```
Target : dependent_files
sample.eva : sample.obj sample.opt sample.cgr          _dependency rule
    186k/p sample;                                     _generating command
    copy      sample.eva c:\myprog\                    _generating command
```

### Dependency rules

Each generation rule starts with a dependency rule. This is in two parts, separated by a colon. The item before the colon is the target, which is the file which MAKE is going to update. The items after the colon are the files on which the target is dependent (also referred to as the source files). In the above example, sample.eva is the target, depending on the files sample.obj, sample.opt, and sample.cgr. There must be no spaces or tabs at the beginning of the dependency rule. There can be more than one target or dependent file, separated by spaces.

The dependency rule means that the target must be updated if it either does not exist, or is older than any of the files on which it depends. As an exception, if there are no files to the right of the colon, the target is always updated. If there are multiple dependency rules in the makefile, by default the first is the final target, so it is best to make the dependency rule for the EVA file the first in the makefile (it is also possible to specify the target explicitly when running MAKE). For the final target, the default target file extension can be set using the .TARGET directive. If a dependency rule is long, it can be broken into a number of lines, by ending all but the last line with a backslash character.

### Example of continued lines

```
sample.eva :                                     sample.obj \
                                                sample.opt \
                                                sample.cgr
                                                186k/p
                                                sample;
sample.obj :                                     sample.asm
                                                m86k sample;
```

### Commands

The commands immediately follow a dependency rule. Each of the command lines must begin with a space or tab character. MAKE uses the existence or not of space or tab characters to distinguish dependency rules from commands. The commands, one on each line, are the DOS commands required to update the target. These commands can include DOS internal commands (such as dir). If a command is long, it can be broken into a number of lines, by ending all but the last line with a backslash character. The commands are passed to DOS by MAKE, so are subject to the line length restriction for DOS (maximum 127 characters).

**Caution:** During execution of MAKE, about 100 KB is required for MAKE itself and work areas. It is therefore possible to run out of memory when by starting make.

---

### Command modifiers

Command modifiers provide more detailed control of command execution. The command modifiers precede the command, and more than one can be attached to a single command.

#### **\_command**

Do not echo this command to the display when executing it. This does not affect output to the display by the command itself. See the related functions, the /S option switch, and .SILENT directive.

---

**Note:** The /S option suppresses command echoing for the whole makefile.  
The .SILENT directive switches the mode on and off through the text of the makefile.

---

#### **-command**

Ignore the command result code. MAKE normally terminates if the result code from a command is other than zero, but if the - modifier is used, MAKE continues regardless of the result code. See the related functions, the /I option switch, and .IGNORE directive.

---

**Note:** The /I option causes result codes to be ignored for the whole makefile.  
The .IGNORE directive switches the mode on and off through the text of the makefile.

---

### Examples of command modifiers

```
sample.eva : sample.obj subr.obj sample.opt sample.cgr
@echo Now creating sample.eva                _Echo progress
l86k/p sample+subr;
sample.obj : sample.asm
-m86k sample;                                _Ignore assembly errors
subr.obj : subr.asm
-m86k subr;                                  _Ignore assembly errors
```

### Phantom targets

By deliberately specifying as a target a file which does not exist, it is possible to force MAKE to execute particular commands. This can be referred to as "phantom target." Obviously, when using a particular name for a phantom target, it is necessary to check that this file does not actually exist in the current directory.

### Example using a phantom target

```
all : copy sample1.eva sample2.eva
sample1.eva : sample1.obj sample1.opt sample1.cgr
             m86k/p sample1;
sample2.eva : sample2.obj sample2.opt sample2.cgr
             m86k/p sample2;
copy : copy sampl?.eva c:\old_prog\
```

In the above makefile, if the "all" target is specified to MAKE, or no target specified, the phantom target all causes both the sampe1.eva and sample2.eva targets to be built, and also, before this, the phantom target "copy" causes sample1.exa and sample2.eva to be copied to the directory c:\old\_prog.

## Macros

A macro allows one character string appearing in the makefile to be replaced by a different character string. Its function is very similar to a "#define" preprocessor statement in C. There are two types of macro: user-defined macros, and built-in macros.

### User macro definition

To define a new macro, use the following syntax.

```
macroname=string
```

Here, macroname can be any combination of alphanumeric characters and underscores, up to a maximum of 255 characters. The characters in the macro name are not case dependent, so for example MacroName and MACRONAME are regarded as the same macro. Another macro can be referenced within macroname, as long as it has already been defined in the makefile.

The right hand string specifies a character string of any length. It must be contained within a single logical line, but can be continued over line breaks by using a backslash character immediately before the line break. It is also possible to specify an empty string of zero length. In this case, when the macro is referenced, since it is replaced by an empty string, this can be used to delete a character string. If the same macro is defined more than once, the latest definition is the one which is used.

### Internal macros and external macros

There are two types of user definition macro: "internal macros", which are defined and referenced in the makefile, and "external macros," which are supported by the MS-DOS shell function using environment variables. The format of the two is the same. By default the internal macros take precedence over external macros, but if the /E option is specified, external macros take precedence.

### Referencing user macros

To reference a macro, enter a dollar sign followed by the macro name in parentheses. If the macro name is a single character, the parentheses can be omitted.

```
$(macroname) or $c
```

If an undefined macro is referenced, it is replaced by an empty string.

### Referencing built-in macros

MAKE provides the following built-in macros for file names.

\$@	Full name of current target file (including path, base, and extension)
\$*	Name of current target file, excluding extension
\$?	List of dependent files newer than target

**Example of macro setting and reference**

```

ASM = m86k                               # LC86000 series assembler
LINK = l86k/p                             # LC86000 series linker

all : sample.eva                          _uses phantom target
sample.obj :                               $*.asm          _references base name of target
                                           $ (ASM) $*;       _references assembler command
sample.eva :                               $*.obj $*.opt $*.cgr  _references target base name
                                           $ (LINK) $*;       _references linker command

```

**Directives**

The following directives can appear in the makefile. Each directive is written on a line with no space (or tab) characters at the start; it must not be within the body of a generation rule.

**.IGNORE: {+|-}**

This switches on and off the mode for ignoring the result codes from programs. When followed by a plus sign, this directive switches to the mode in which the result codes are ignored; when followed by a minus sign, it switches to the mode in which the result codes affect MAKE execution. By default, if a result code is other than zero, MAKE terminates. To ignore the result code from a single command only, use the minus sign modifier. To ignore result codes for the whole makefile, use the /I option.

**.SILENT: {+|-}**

This switches on and off the mode for echoing programs run from the makefile. When followed by a plus sign, this directive switches to the mode in which commands are not echoed; when followed by a minus sign, it switches to the mode in which commands are echoed. By default, commands are echoed. Suppress the echo from a single command only, use the at sign modifier. To ignore suppress echoing for the whole makefile, use the /S option.

**.DEFAULT:**

When a generation rule in the makefile consists of a dependency rule with no following commands, MAKE uses the implicit rules to generate commands. If there are no creation rules, a default set of commands can be supplied with the .DEFAULT directive on a line followed by the commands.

**.DEFAULT**

commands

**.TARGET: suffixes**

The final target to be built can be specified to MAKE on the command line, and if this is omitted, the target of the first generation rule in the makefile is built, but the .TARGET directive specifies an extension for the default final target, so that the extension of a file to be built can be specified for the final target. A suffix is a period followed by up to three characters. More than one suffix can be specified by separating them with spaces.

# Implicit Rules

The implicit rules define how the general way to make a file of one extension from a file of another extension. MAKE follows these implicit rules to derive the commands needed to update a target, from the target dependency rules. Using implicit rules generally makes writing the makefile simpler. The implicit rules can be included in the makefile, or written in the makerule file MAKERULE.DEF.

For each of the source files in a generation rule, MAKE checks whether there is generation rule with that file as target, and if not it uses an implicit rule. The conditions for using an implicit rule are thus as follows:

- 1) There must be no dependency rule for the file in question.
- 2) There must be a generation rule for making the file.
- 3) The file or files required for making the file must exist.

If these conditions are met, MAKE adds generation rules as follows.

- 1) If a generation rule does not exist, it is added.
- 2) `basename.sss` is added as the source file (sss: source file extension)
- 3) The commands from the (implicit) generation rule are used as the commands for creating the target.

## Makerule file

The makerule file MAKERULE.DEF contains the rules that MAKE uses to create implicit rules, in the following format.

```
.sss.ttt:  
commands
```

The first line specifies two file extensions. The first, "sss," is the extension of the source file, and the second, "ttt," is the extension of the target file. The extensions are not case-dependent. The first period, before "sss," must come at the very beginning of a line. The following lines are the commands, written as in the makefile, for creating the target. For example, to create an object file "basename.obj" from an assembly language source "basename.asm," the rule is ".asm.obj."



**Example makerule file**

```
# *****
# ***
# ***      Implicit rules for EVA86000 utility make.      ***
# ***      Definition for M86K                            ***
# ***
# *****
ASM = m86k

.ASM.OBJ:
                                $ (ASM) $*;

.TARGET: .EVA .HEX __default final target
.DEFAULT:
                                @echo
-----
                                @echo ??? Undefined build commands ???
                                @echo
-----
# end of makerule.def
```

**Example makefile using a makerule file**

```
ASM = m86k                                # LC86000 series assembler
LINK = l86k/p                             # LC86000 series linker

all : sample.eva

sample.obj : $*.asm

sample.eva : $*.obj $*.opt $*.cgr         _ assembler command omitted
                                $ (LINK) $*;
```



# Assembler Syntax

---

Each line of the source file is a character string of up to 511 characters (including the terminating CR and LF). Except for symbols defined in the source program (labels, macros, and so on), uppercase and lowercase letters are not distinguished. For example, "Nop" and "nop" are both recognized as the mnemonic for the NOP instruction. By specifying the -I assembler option, it is possible to remove the case distinctions for labels and other symbols as well.

## Statements

Statements consist of the instruction mnemonics and operands which define the object code to be created by the assembly process, and comments. One line of source code corresponds to one instruction mnemonic. A statement is not allowed to be continued over more than one line. Each statement comprises the following four fields.

```
[label:] [operation] [operand] [;comment]
```

Field	Purpose
label	Identifies the location of this statement, so that it can be referenced from another statement. It must always be followed by the colon.
operation	Specifies the function of the statement.
operand	Specifies the operand (or operands) on which the function operates.
comment	This is for explanatory purposes, and does not directly affect the result of assembly.

---

**Caution:** The square brackets [ ] identify Items which can be omitted.

---

## Label and Symbol Names

Label and symbol names consist of character strings of any (nonzero) length, but only the first 32 characters are used to distinguish names. The following characters can be used:

A to Z, a to z, 0 to 9, \$, ?, \_(underscore), @, . (period)

Label and symbol names must begin with a letter, underscore, period or '@.' If the -i assembler option is specified, uppercase and lowercase letter are regarded as the same. Note that a label must be followed by a colon.

## Comments

Comments are delimited by a semicolon, and extend to the end of the line.

## Operators

The following table lists the operators which can be used in M86K assembly language, and their order of precedence. For operators such as NOT whose names consist of letters, case is not distinguished, and thus "NOT" and "not" are both the same operator.

Operator	Meaning	Precedence order
NOT	One's complement logical not	1
HIGH	High order byte	
LOW	Low order byte	
*	Multiplication	2
/	Division	
MOD	Modulo (remainder)	
+	Addition	3
-	Subtraction	
SHR	Shift right	4
SHL	Shift left	
LAND	Logical AND	5
LOR	Logical (inclusive) OR	
LXOR	Logical exclusive OR	
EQ	Equals	6
NE	Not equals	
LT	Less than	
LE	Less than or equals	
GT	Greater than	
GE	Greater than or equals	

## Numeric Constants

M86K allows numeric constants to be written in binary, octal, decimal, or hexadecimal. Constants can be written with an explicit indication of the radix, or base, as for example in "123H," or with the default radix defined by the RADIX pseudoinstruction. Thus a constant "123," for example, with no explicit radix is interpreted according to the specification of the RADIX pseudoinstruction. By default, if there is no RADIX pseudoinstruction, such numbers are taken as decimal.

However they are written, constants are handled internally by the assembler as 32-bit values. When the final result of a numeric expression is evaluated and stored as immediate data as the operand of an instruction, only the number of bits required for the operand are stored, and any more significant bits are discarded.

**Table 2.35 Notation of constants with an explicit radix**

Radix	Format	Examples
2	'%' followed by one or more digits 0, 1	%01111011 %1111111 %0000010000000000
	One or more digits 0, 1 followed by 'B' *	01111011B 1111111B 0000010000000000B
	One or more digits 0, 1 followed by '.B' *	01111011.B 1111111.B 0000010000000000.B
8	One or more digits 0 to 7 followed by '.O	'273.O 377.O 2000.O
10	One or more digits 0 to 9 followed by '.D	'123.D 255.D 1024.D
16	'\$' followed by one or more digits 0 to 9, a to f, or A to F	\$7B \$FF \$0400
	One or more digits 0 to 9, a to f, or A to F, followed by 'H' (must start with 0 to 9)	7BH OFFH 0400H
	One or more digits 0 to 9, a to f, or A to F, followed by '.H' (must start with 0 to 9)	7B.H OFF.H 0400.H

**Note:** The radix letters B, O, D, and H can be uppercase or lowercase. This is not affected by the assembler -i option.

**Caution:** These formats are affected by the RADIX setting. See the table below.

**Table 2.36 Interpretation of numeric constants without explicit radix notation**

Format	Example	Values for each RADIX setting (in decimal)			
		2	8	10	16
One or more 0, 1	0101	510	6510	10110	25710
One or more 0 to 7	123	Error	8310	12310	29110
One or more 0 to 9	789	Error	Error	789110	192910
One or more 0, 1 followed by 'B'	'101B	510	510	510	412310
One or more digits 0 to 9, a to f, or A to F, starting with 0 to 9	OFF	Error	Error	Error	25510

## Character Constants

A character enclosed in single quotes (') is treated as a character constant. A character constant is a type of numeric constant, with the value of the ASCII codes of the specified characters. In addition to all printable ASCII characters, the following codes can be used to enter other characters. If more than one character is enclosed in the quotes, this is not a character constant but a character string constant see Section , "Character String Constants,".

**Table 2.37** *Codes for use in character constants and character string constants*

Notation	Hexadecimal value	Character name
\n	0A	Linefeed
\r	0D	Carriage return
\t	09	Horizontal tab
\b	08	Backspace
\f	0C	Form feed
\	"22	Double quote
\	'27	Single quote
\\	5C	Backslash
\ooo		Octal value ooo
\xhh		Hexadecimal value hh

**Example 1:** ADD \_'A'

**Example 2:** DB 'A', '\012', 'C'

**Example 3:** DB 'ABC'

---

**Caution:** In example 3, 'ABC' is a character string constant, and is therefore an error as the operand for DB.

---

## Character String Constants

One or more characters enclosed in double quotation marks ("), or two or more characters enclosed in single quotation marks (') are treated as a character string constant. A character string constant can be used as the operand of a DC or .PRINTX pseudoinstruction. Within a character string, any printable ASCII characters can be used, and also the codes listed in Section , "Character Constants,".

### Example

```
DC  "This is a sample string with special codes \007\r\n"
```

## Special Symbols

In an operand, an asterisk represents the address of the current location.

### Example 1

The following represents the address 6 bytes before the current address.

```
BR  *-6
```

### Example 2

The following represents the address 12 bytes after the current address.

```
BR  *+12
```





# Assembler Pseudoinstructions

A pseudoinstruction differs from an ordinary instruction (such as ADD or MOV in the LC86K instruction set); it gives directives or definitions to the assembler, and a pseudoinstruction of itself does not generate a machine instruction. (This does not apply to JMPO and other optimization pseudoinstructions, or to the CHANGE pseudoinstruction.) Pseudoinstructions are often used in combination with ordinary instructions.

Group	Pseudoinstruction	Function
Linking control	ORG WORLD CSEG DSEG END PUBLIC EXTERN OTHER_SIDE_SYMBOL	Specify origin Select the ROM to hold code Declare the beginning of a code segment Declare the beginning of a data segment End program Declare public symbol Declare external symbol Declare CHANGE instruction jump label
Symbol definitions	EQU SET	Assign a fixed value Assign temporary value
Data definitions	DB DW DC DS	Define byte data Define word data Define character string data Define byte area
Macro control	MACRO REPT IRP IRPC ENDM EXITM LOCAL	Define macro Repeat macro Iteration macro Character string macro End macro definition End macro expansion Define local label

Conditional assembly	IFDEF IFNDEF IFB IFNB IFE IFNE IFIDN IFDIF ELSE ENDIF .PRINTX .LIST .XLIST .MACRO .XMACRO .IF .XIF	Assemble if defined Assemble if undefined Assemble if operand empty Assemble if operand nonempty Assemble if zero Assemble if nonzero Assemble if identical Assemble if different Else case of conditional assembly End conditional assembly Display message during assembly Resume listing Suppress listing List macro expansions End macro expansion listing List skipped statements in conditional assembly End listing of skipped statements Assemble if operand empty
Miscellaneous	INCLUDE TITLE PAGE CHIP COMMENT WIDTH BANK CHANGE RADIX	Include file Set listing title New page Specify chip for assembly Add comment to object file Specify columns in listing file Specify RAM bank Jump between flash memory and ROM Specify default radix

Optimization	JMPO	Optimized JMP instruction
	BRO	Optimized BR instruction
	CALLO	Optimized CALL instruction
	BZO	BZ instruction guaranteeing no address error
		BNZ instruction guaranteeing no address error
	BNZO	BP instruction guaranteeing no address error
		BPC instruction guaranteeing no address error
	BPO	BN instruction guaranteeing no address error
		DBNZ instruction guaranteeing no address error
	BPCO	BE instruction guaranteeing no address error
		BNE instruction guaranteeing no address error
	BNO	Optimized BR instruction
		Optimized CALL instruction
	DBNZO	BZ instruction guaranteeing no address error
		BNZ instruction guaranteeing no address error
	BEO	BP instruction guaranteeing no address error
		BPC instruction guaranteeing no address error
	BNEO	BN instruction guaranteeing no address error
		DBNZ instruction guaranteeing no address error

**ORG****Specify origin****Syntax****ORG expression****Description**

The ORG pseudoinstruction specifies the start address in program memory (flash memory) as expression. Expression must be a numeric constant, or an expression which can be evaluated at assembly time.

## Example

```
page:      1 <org.ASM>
ERR SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                               ;a sample program for ORG
0002                               chip   lc866032
0003                               extern wait1s
0004                               dseg
0005      D 0000      min1:  ds    1
0006      D 0001      min0:  ds    1
0007                               cseg
0008                               org    0h
0009      C 0000 6201'  label1: inc   min0
0010      C 0002 0201'          ld    min0
0011      C 0004 A13C          sub   #60
0012      C 0006 900311       bzo   label2
0012      C 0009 F600
0013      C 000B 210200'      jmpf  label3
0014                               org    100h
0015      C 0100 6200'  label2: inc   min1
0016      C 0102 220100'      mov   #00,min0
0017      C 0105 210200'      jmpf  label3
0018                               org    200h
0019      C 0200 100000' label3: CALLr wait1s
0020      C 0203 210000'      jmpf  label1
0021                               end
```

## WORLD

### Select the ROM to hold code

### Syntax

### WORLD selection

### Description

This specifies the ROM which will hold the assembled code. This pseudoinstruction is only valid when the target chip is the LC86800 series. There are three values which can be specified for selection.

INTERNAL	Store in the on-chip ROM.
EXTERNAL	Store in flash memory bank 0.
EXTERNAL_DATA	Store in flash memory bank 1.

**Caution:** For Visual Memory, always specify EXTERNAL. Other specifications may lead to data corruption or misoperation.

---

If there is more than one WORLD pseudoinstruction in a single file, an error results. For chips other than the chips other than the LC86800 series, if a value other than INTERNAL is selected for the WORLD pseudoinstruction, an error results.

### **CSEG**

**Declare the beginning of a code segment**

### **Syntax**

**CSEG mode**

### **Description**

This indicates to the assembler the beginning of a segment holding program code. When mode is not specified or is INBLOCK, the segment is aligned within 4K boundaries. If the mode is FREE, this indicates that the segment can be located regardless of 4K boundaries.

## Example

```

page:      1 <cseg.ASM>
ERR SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                               ; a sample program for CSEG
0002                               chip   lc864024
0003                               extern wait1s
0004                               dseg
0005      D 0000      min1:  ds    1
0006      D 0001      min0:  ds    1
0007                               cseg  inblock
0008      C 0000 6201' label1: inc   min0
0009      C 0002 0201'      ld    min0
0010      C 0004 A13C      sub   #60
0011      C 0006 900311    bzo   label2
0011      C 0009 0000
0012      C 000B 210000'   jmpf  label3
0013                               cseg  free
0014      c 0000 6200' label2: inc   min1
0015      c 0002 220100'   mov   #00,min0
0016      c 0005 210000'   jmpf  label3
0017                               cseg
0018      C 0000 100000' label3: CALLr wait1s
0019      C 0003 210000'   jmpf  label1
0020                               end

```

Local address is reset to zero at the beginning of each segment.

Independent segments

### DSEG

Declare the beginning of a data segment

### Syntax

### DSEG

### Description

This indicates to the assembler the beginning of a segment holding data.

**Caution:** Data segments are copied into RAM. It is not possible to open a data segment in flash memory.

**Example**

```

page:      1 <dseg.ASM>
ERR SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                ; a sample program for CSEG
0002                chip  lc864024
0003                extern wait1s
0004                cseg  inblock
0005 C 0000 6201' label1: inc  min0
0006 C 0002 0201'      ld   min0
0007 C 0004 A13C      sub  #60
0008 C 0006 900311    bzo  label2
0008 C 0009 0000
0009 C 000B 210000'   jmpf label3
0010                cseg  free
0011 c 0000 6200' label2: inc  min1
0012 c 0002 220100'   mov  #00,min0
0013 c 0005 210000'   jmpf label3
0014                cseg
0015 C 0000 100000' label3: CALLr wait1s
0016 C 0003 210000'   jmpf label1
0017
0018                dseg
0019 D 0000      min1: ds   1
0020 D 0001      min0: ds   1
0021                end
    
```

**END**

**End program**

**Syntax**

**END**

**Description**

This indicates the end of the source program. When the assembler encounters this instruction, it ends the pass currently being executed, so any text beyond this point, even if valid statements, is ignored.

## Example

```
; a sample program for END
chip    lc866032
cseg
mov     #20h, 01h
mov     #10h, 00h
ld      00h
add     0fh
end
inc     00h
inc     01h
ld      01h
```

↓  
All text after END is ignored

```
page:    1 <end.ASM>
ERR SEQ.  S LOC. OBJ.  SOURCE STATEMENTS
0001                ; a sample program for END
0002                chip    lc866032
0003                cseg
0004      C 0000 220120    mov     #20h, 01h
0005      C 0003 220010    mov     #10h, 00h
0006      C 0006 0200      ld      00h
0007      C 0008 820F      add     0fh
0008                end
```

## PUBLIC

### Declare public symbol

### Syntax

**PUBLIC**    **symbol {, symbol}**

### Description

The PUBLIC pseudoinstruction declares that a symbol defined in the program can be referenced from other source files.



## Example

```

page:      1 <extern.ASM>
ERR SEQ.   S LOC. OBJ.   SOURCE STATEMENTS
0001                ; a sample program for EXTERN
0002                chip   lc866032
0003                extern label1, label2
0004
0005                cseg  inblock
0006 C 0000 200000' CALLf  label1
0007
0008 C 0003 200000'start: CALLf  label2
0009 C 0006 0303      ld     c
0010 C 0008 90F9      bnz   start
0011
0012 C 000A A300      sub   a
0013
0014                end
  
```

To reference a symbol defined in another file, it must be declared EXTERN.

To allow a symbol in this file to be referenced from another file, it must be declared PUBLIC.

```

ERR SEQ.   S LOC. OBJ.   SOURCE STATEMENTS
0001                ; a sample program for PUBLIC
0002                chip   lc866032
0003                public label1, label2
0004
0005                cseg  inblock
0006 C 0000 220000' label1: mov  #00, data1
0007 C 0003 23033C      mov  #60, c
0008 C 0006 A0          ret
0009
0010 C 0007 6200' label2: inc  data1
0011 C 0009 0200'      ld   data1
0012 C 000B 410A05      bne  #10, label3
0013 C 000E 220000'      mov  #00, data1
0014 C 0011 6201'      inc  data2
0015
0016 C 0013 7303 label3: dec  c
0017 C 0015 A0          ret
0018
0019                dseg
0020 D 0000      data1: ds  1
0021 D 0001      data2: ds  1
0022
0023                end
  
```

**Caution:** To reference a symbol defined in another source file, it must be declared EXTERN.  
 To allow a symbol in this file to be referenced from another file, it must be declared PUBLIC.

```
page:1 <public.ASM>
ERR SEQ.      S LOC. OBJ.      SOURCE STATEMENTS
0001          ; sample program for PUBLIC
0002          chip      lc866032
0003          public   label1, label2
0004
0005          cseg      inblock
0006          C 0000 220000'  label1:  mov    #00, data1
0007          C 0003 23033C    mov    #60, c
0008          C 0006 A0        ret
0009
0010          C 0007 6200'    label2:  inc    data1
0011          C 0009 0200'    ld      data1
0012          C 000B 410A05    bne    #10, label3
0013          C 000E 220000'    mov    #00, data1
0014          C 0011 6201'    inc    data2
0015
0016          C 0013 7303    label3:  dec    c
0017          C 0015 A0        ret
0018
0019          dseg
0020          D 0000    data1:  ds    1
0021          D 0001    data2:  ds    1
0022
0023          end
```

---

**Note:** The combination of PUBLIC and EXTERN declarations allows a symbol to be referenced even when it is defined in another file.

---

### EXTERN

#### Declare external symbol

#### Syntax

**EXTERN** [segment: ] symbol {, [segment: ] symbol}

#### Description

The EXTERN pseudoinstruction is used when a symbol or symbols are defined in other source program files. The optional segment parameter is either CSEG or DSEG, indicating the segment type. If this is not specified, a code segment, CSEG, is the default.

---

**Reference:** For examples see the previous item "PUBLIC - Declare public symbol."

---

### OTHER\_SIDE\_SYMBOL

#### Declare CHANGE instruction jump label

### **Syntax**

**OTHER\_SIDE\_SYMBOLlabel {, label}**

### **Description**

This declares an address label which can be specified as the operand of a CHANGE instruction, which in the LC86800 series is used for switching between ROM and flash memory. The label declared is a type of external symbol, but one difference is that in a source file of code to be stored in ROM, a label is declared in flash memory (or in ROM in a source file of code to be stored in flash memory). This pseudoinstruction is only valid for the LC86800 series, and in other cases an error results.

---

**Reference:** For examples, see under "CHANGE - Jump between flash memory and ROM in this chapter."

---

### **EQU**

**Assign a fixed value**

### **Syntax**

**Symbolname EQU expression**

### **Description**

The EQU pseudoinstruction assigns the value expression to symbolname. A symbol defined with the EQU pseudoinstruction cannot be redefined. Used appropriately, the EQU pseudoinstruction can aid the visual identification of constant data, and improve maintenance efficiency.

## Example

When the defined value can be computed it appears here (hexadecimal).

No colon between the defined symbol and the "EQU"

Any expression can be written.

```
page:      1 <equ.ASM>
ERR SEQ.   S LOC. OBJ. SOURCE STATEMENTS
0001      ; a sample program for EQU
0002      chip      lc866032
0003
0004      00000064 loop_max      equ      100
0005      00000001 mode_a       equ      1
0006      00000002 mode_b       equ      2
0007      00000003 mode_c       equ      3
0008
0009      cseg      inblock
0010      C 0000 220000'      mov      #00, loop_ctr
0011
0012      C 0003 230201 label1: mov      #mode_a, b
0013      C 0006 0818'        CALL    subl
0014      C 0008 230202      mov      #mode_b, b
0015      C 000B 0818'        CALL    subl
0016      C 000D 230303      mov      #mode_c, c
0017      C 0010 6200'        inc      loop_ctr
0018      C 0012 0200'        ld      loop_ctr
0019      C 0014 4164EC      bne     #loop_max, label1
0020      C 0017 A0          ret
0021
0022      C 0018 0302      subj:  ld      b
0023      C 001A 310107      be      #mode_a, suj10
0024      C 001D 310208      be      #mode_b, suj11
0025      C 0020 310309      be      #mode_c, suj12
0026      C 0023 A0          subj0: ret
0027
0028      C 0024 1201'      suj10: st      data_a
0029      C 0026 01FB      suj0:  br      suj0
0030      C 0028 1202'      suj11: st      data_b
0031      C 002A 01F7      suj0:  br      suj0
0032      C 002C 1203'      suj12: st      data_c
0033      C 002E 01F3      suj0:  br      suj0
0034
0035      dseg
0036      D 0000      loop_ctr:  ds      1
0037      D 0001      data_a:  ds      1
0038      D 0002      data_b:  ds      1
0039      D 0003      data_c:  ds      1
0040
0041      end
```

## SET

Assign temporary value

## Syntax

Symbolname SET expression

## Description

The SET pseudoinstruction assigns the value expression to symbolname. A symbol defined with the SET pseudoinstruction can be redefined by a subsequent SET. However, a symbol set with this pseudoinstruction cannot be the subject of a PUBLIC declaration, nor can it be redefined with EQU.

## Example

```
page:      1 <set.ASM>
ERR SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                ; a sample program for SEY
0002                chip  1c866032  When the defined value can be
0003                cseg  inblock   computed it appears here
0004                (hexadecimal).
0005      00000000 dd  set  0
0006                No colon between the defined
0007      C 0000 220000'  mov  #dd,zz+dd  symbol and the "SET"
0008
0009      C 0003 6300    inc  a
0010      C 0005 6302    inc  b
0011
0012      00000001 dd  set  dd+1
0013
0014      C 0007 220101'  mov  #dd,zz+dd
0015
0016      C 000A 7300    dec  a
0017      C 000C 7302    dec  b
0018
0019                dseg
0020      D 0000        zz:  ds   2
0021
0022                end
```

*Any expression can be written, including the symbol currently being set.*

## DB

Define byte data

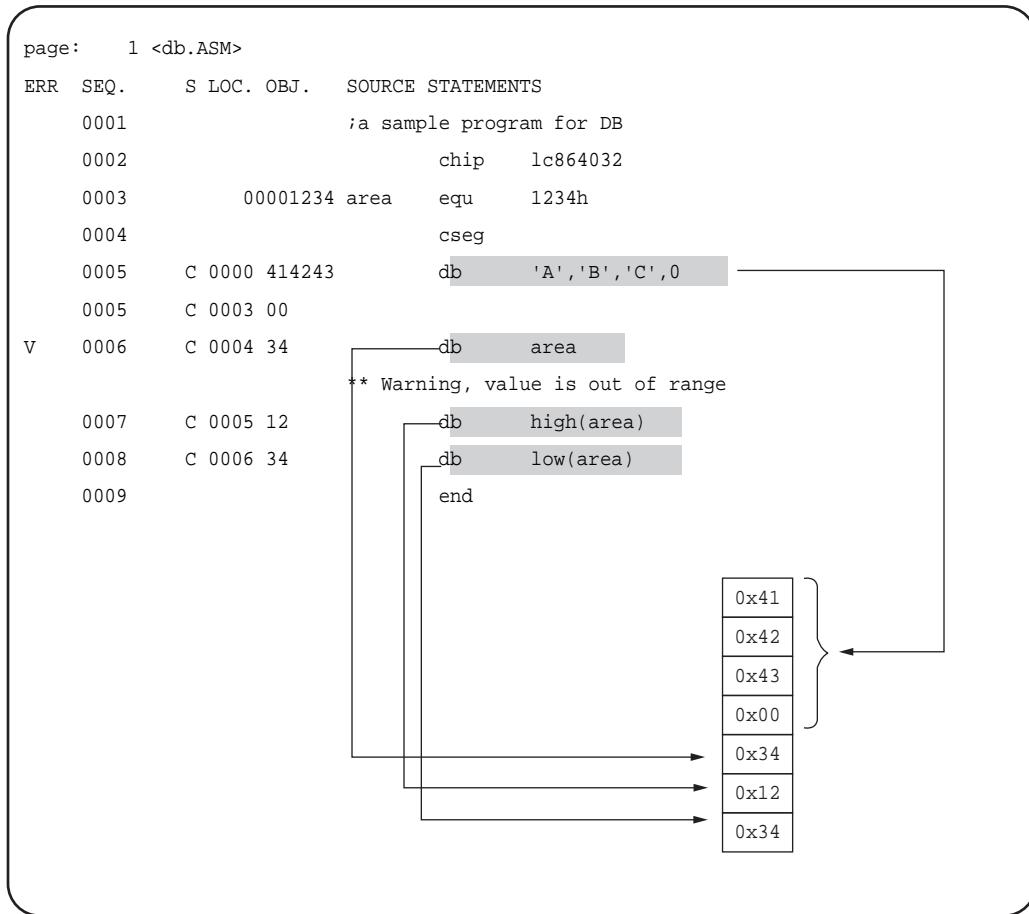
## Syntax

Labelname DB expression {, expression}

## Description

The DB pseudoinstruction stores the 8-bit data value or values represented by expression in program memory (ROM). Any number of operands may be specified, separated by commas. When two or more operands are specified, they are evaluated in order left to right, and stored in successive addresses. If there are two commas with nothing between them, this is interpreted as a zero value.

## Example



In the above example, because the "db area" statement references the symbol "area," which is a 16-bit value, at assembly time a warning level message, "Value is out of range," is generated. The bottom eight bits of the value are used in the object code.

## DW

### Define word data

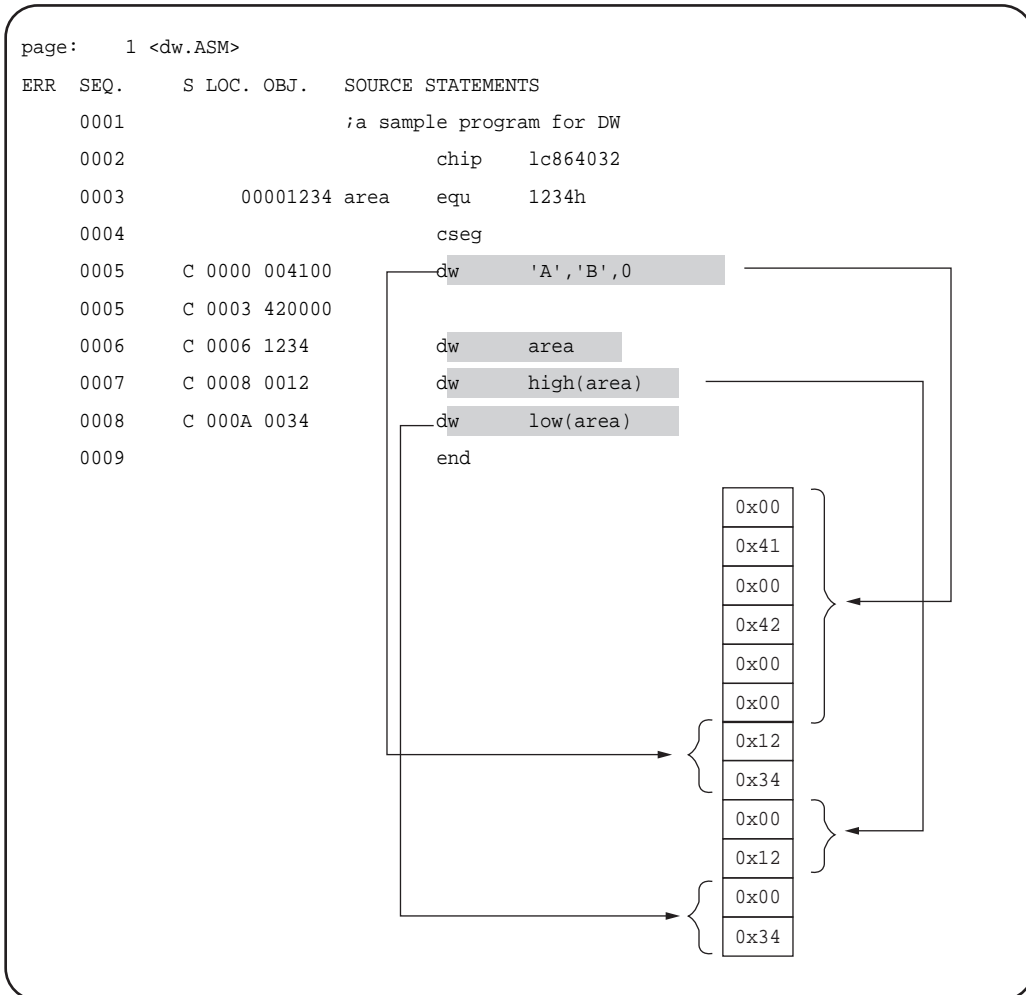
### Syntax

**labelname DW expression {, expression}**

**Description**

The DW pseudoinstruction stores the 16-bit data value or values represented by expression in program memory (ROM). The more significant byte is stored first, and the less significant byte at the address one higher. Any number of operands may be specified, separated by commas. When two or more operands are specified, they are stored in successive addresses. If there are two commas with nothing between them, this is interpreted as a zero value.

**Example**



If the DW pseudoinstruction is used to define 8-bit values, the upper 8 bits of the 16-bit result are always 0.

**DC**

**Define character string data**

**Syntax**

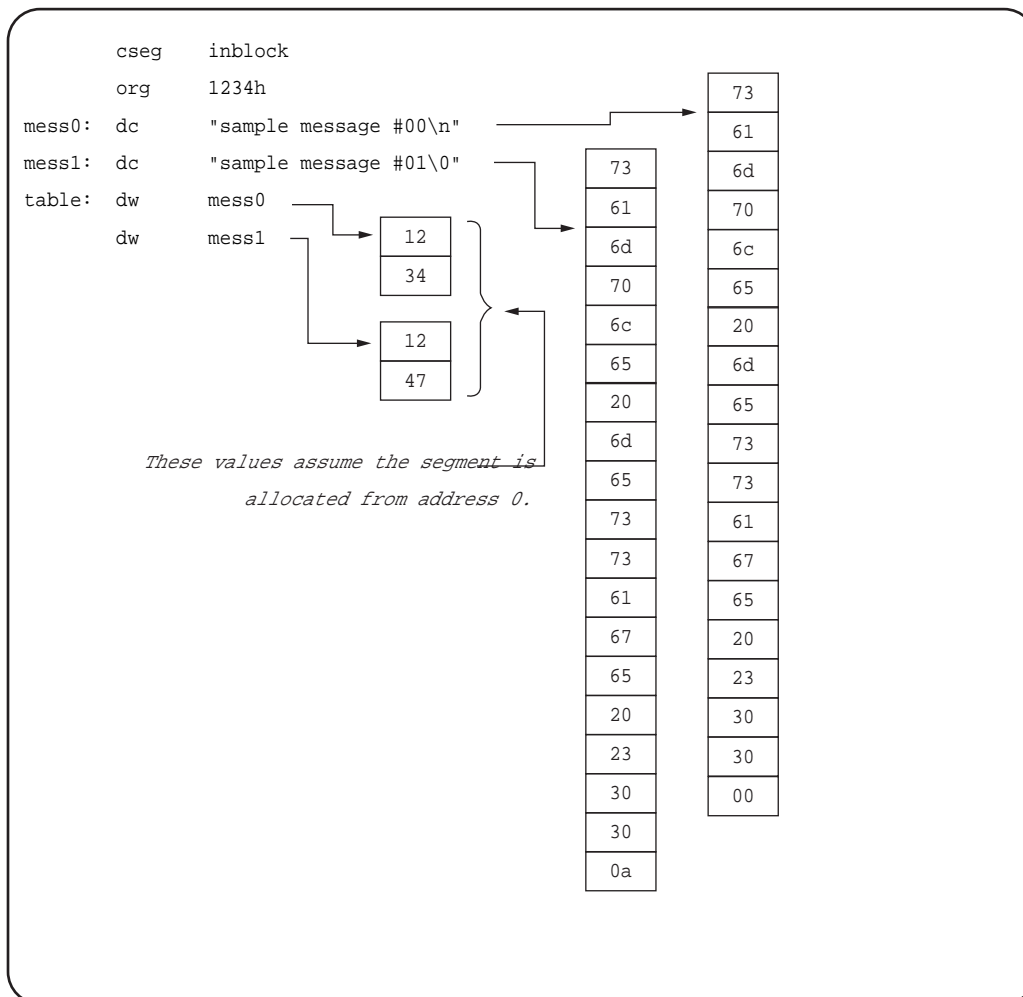
**labelname DC "string"**

## Description

This stores the ASCII codes of string (a character string constant) in sequence in program memory (ROM).

**Reference:** For details of character string constants, see Section 20.7, "Character String Constants."

## Example



## DS

Define byte area

## Syntax

labelname DS absolute\_expression

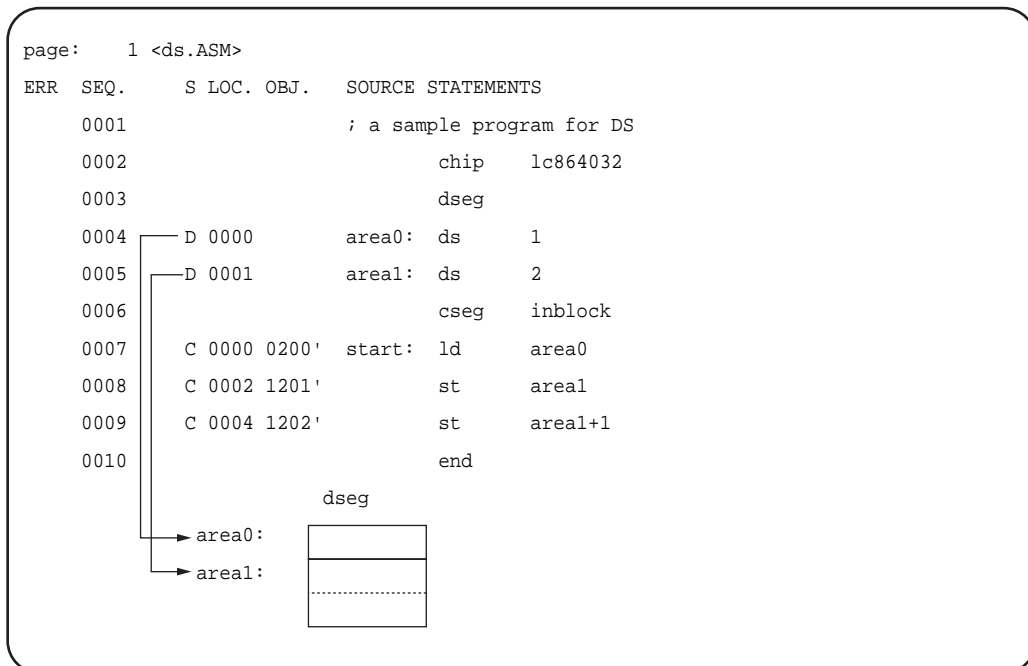


## Description

The DS pseudoinstruction allocates any area of data memory (RAM) of the number of bytes specified by `absolute_expression`. The `absolute_expression` must have a value completely determined at assembly time. This pseudoinstruction can only be used after a DSEG pseudoinstruction.

**Caution:** A DS pseudoinstruction can only be used to allocate RAM (a data segment). It cannot be used for flash memory. Use DB or DW statements instead.

## Example



The example above defines a 1-byte area named `area0` that is immediately followed by a 2-byte area named `area1`.

## MACRO

### Define macro

### Syntax

**name**      **MACRO** **parameter** {, **parameter**}

### Description

The MACRO pseudoinstruction defines a macro. The statements from the MACRO pseudoinstruction to the following ENDM pseudoinstruction form the body of the macro. The parameter name is the name by which the macro can be called, which is replaced by the body of the macro, and is therefore mandatory. The formal parameter list specified by parameter is optional, depending on the macro being defined.

**Caution:** When calling one macro from within another, or when using a pseudoinstruction such as IFB which requires angle brackets (<>), a sufficient depth of angle brackets to correspond to the nesting level is required. See the section "EXITM – End macro expansion" in this chapter.

---

## Example

```
_push macro
  push  acc
  push  c
  push  b
endm
```

← Push acc, c, and b onto the stack.

```
_pop macro
  pop   b
  pop   c
  pop   acc
endm
```

← Pop b, c, and acc off the stack.

```
_shl macro count
  ifne count
    rept count
      rolc
    endm
  else
    .printx "logical shift count is zero !!\007"
  endif
endm
```

↑ Generates code to left shift by the number of positions shown by the parameter.  
However, if the parameter is zero, generates no code.

```
cseg
start: _push
       _shl 0
       _shl 2
       _shl 1
```

← Format of source program.

```
0027          start:  _push
0027+1 C 0000 6100    push  acc
0027+2 C 0002 6103    push  c
0027+3 C 0004 6102    push  b
0028          _shl  0
0028+1          ifne  0
0028+2          rept  0
0028+3          rolc
0028+4          endm
0028+5          else
0028+6          .printx "logical shift count is ze
0028+7          endif
0029          _shl  2
0029+1          ifne  2
0029+2          rept  2
0029+4          endm
0029+1 C 0006 F0      rolc
0029+2 C 0007 F0      rolc
0029+5          else
0029+6          .printx "logical shift count is ze
0029+7          endif
0030          _shl  1
0030+1          ifne  1
0030+2          rept  1
0030+4          endm
0030+1 C 0008 F0      rolc
0030+5          else
0030+6          .printx "logical shift count is ze
0030+7          endif
0031          _pop
0031+1 C 0009 7102    pop   b
0031+2 C 000B 7103    pop   c
```

### REPT

#### Repeat macro

#### Syntax

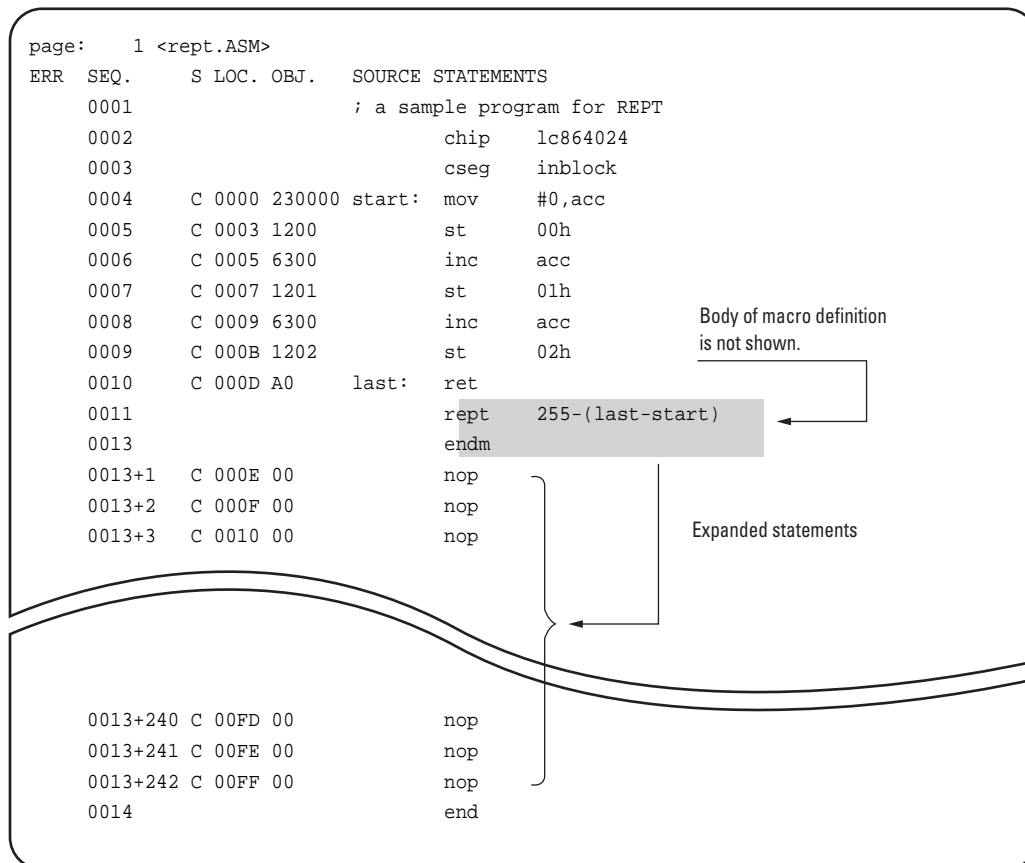
#### REPT count

#### Description

The REPT pseudoinstruction repeats the statements up to the ENDM instruction, generating the number of copies specified by count. This value can be any integer from 1 to 65535.

## Example

In the following example, the area not occupied by the program is filled with NOP codes (for a 256-byte boundary).



## IRP

### Iteration macro

### Syntax

**IRP** parameter, argument {, argument}...

### Description

The IRP pseudoinstruction repeats the statements up to the ENDM instruction, generating one copy for each argument specified. In each copy, parameter is replaced by the corresponding argument.

**Example**

```
_push macro
    irp    reg_name,acc,b,psw,c
        push    reg_name
    endm
endm
_pop macro
    irp    reg_name,c,psw,b,acc
        push    reg_name
    endm
endm
```

```
0016
0017                _push
0017+1              irp    reg_name,acc,b,psw,c
0017+3              endm
0017+1 C 0000 6100    push    acc
0017+2 C 0002 6102    push    b
0017+3 C 0004 6101    push    psw
0017+4 C 0006 6103    push    c
0018                _pop
0018+1              irp    reg_name,c,psw,b,acc
0018+3              endm
0018+1 C 0008 6103    push    c
0018+2 C 000A 6101    push    psw
0018+3 C 000C 6102    push    b
0018+4 C 000E 6100    push    acc
```

**IRPC****Character string macro****Syntax****IRPC** parameter, string

## Description

The IRPC pseudoinstruction repeats the statements up to the ENDM instruction, generating one copy for each character in string. As distinct from a character string constant, string is not enclosed in quotation marks. Codes beginning with a backslash cannot be used. In each copy, parameter is replaced by the corresponding character in string.

## Example

```
; a sample program for IRPC
    chip    lc866032
    dseg
    irpc    x,01234567
buf&x:   ds    2
    endm
    end
```

The formal parameter is replaced by successive characters from the argument string.  
The ampersand delimits the formal parameter when it appears within an identifier.

```
page:      1 <irpc.ASM>
ERR  SEQ.   S LOC. OBJ.   SOURCE STATEMENTS
0001                                ; a sample program for IRPC
0002                                chip    lc866032
0003                                dseg
0004                                irpc    x,01234567
0006                                endm
0006+1  D 0000   buf0:    ds    2
0006+2  D 0002   buf1:    ds    2
0006+3  D 0004   buf2:    ds    2
0006+4  D 0006   buf3:    ds    2
0006+5  D 0008   buf4:    ds    2
0006+6  D 000A   buf5:    ds    2
0006+7  D 000C   buf6:    ds    2
0006+8  D 000E   buf7:    ds    2
```

## ENDM

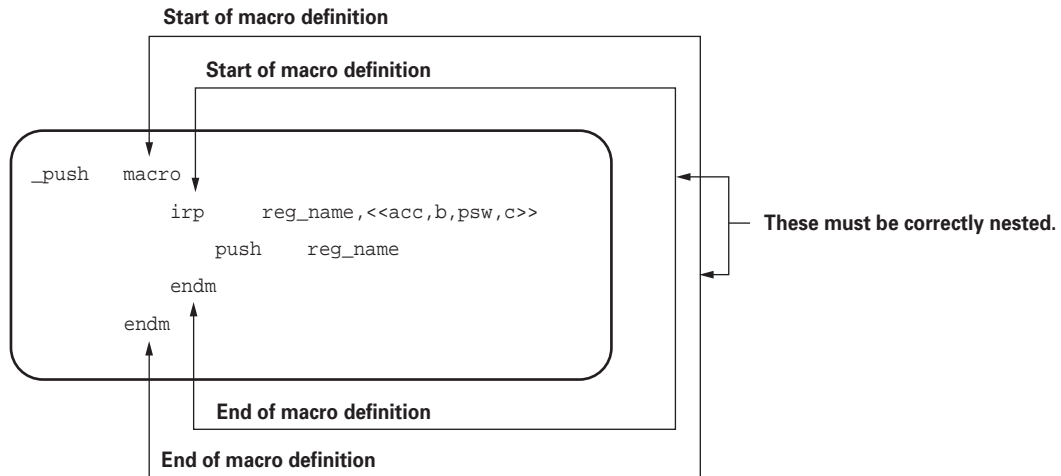
### End macro definition

## Syntax

## ENDM

**Description**

This marks the end of a macro definition.

**Example****EXITM****End macro expansion****Syntax****EXITM****Description**

The EXITM pseudoinstruction ends expansion of a macro. In combination with conditional assembly pseudoinstructions, this can be used to create different forms of expansion of a macro depending on the arguments supplied.

## Example

The angle brackets must be double, because one layer is removed in each macro expansion.

```

page:      1 <exitm.ASM>
ERR  SEQ.      S LOC. OBJ.  SOURCE STATEMENTS
0001          ; a sample program for EXITM
0002          chip      LC866032
0003  rpush   macro   a1,a2,a3,a4
0004          ifb      <<a1>>
0005          .printx "not enough argument"
0006          exitm
0007          endif
0008          ifnb     <<a2>>
0009          push    a1
0010          push    a2
0011          push    a3
0012          push    a4
0013          endif
0014          endm
0015          cseg     inblock
0016          rpush   acc,b,psw,c
0016+1        ifb      <acc>
0016+2        .printx "not enough argument"
0016+3        exitm
0016+4        endif
0016+5        ifnb     <b>
0016+6  C 0000 6100          push    acc
0016+7  C 0002 6102          push    b
0016+8  C 0004 6101          push    psw
0016+9  C 0006 6103          push    c
0016+10       endif
0017          rpush
0017+1        ifb      <>
0017+2        .printx "not enough argument"
0017+3        exitm
0018          end
    
```

When the first argument is applied, this section is assembled.

Since there is no second argument, this section is assembled, and the expansion terminates at EXITM.

## LOCAL

### Define local label

### Syntax

**LOCAL** name {, name}

### Description

The LOCAL pseudoinstruction declares a label which can be used internally to the body of the macro. During macro expansion, the name declared in the LOCAL pseudoinstruction is replaced by the assembler with a unique identifier to avoid name conflicts.



## Example

```

; sample program for LOCAL
                                chip          1c864008
b_ne                             macro        val,dst
                                local         skip
                                be            val,skip
                                bro           dst

skip:

                                endm

cseg

                                b_ne         #0, over
org                               200h
over:                             b_ne         #0, under
                                nop
under:                             nop
                                end

```

In the above example, the BRO pseudoinstruction is used to define the B\_NE macro which generates different instructions depending on the destination of a jump; this is then used in the example. The following is the result of assembly.

```

page:      1 <local.ASM>
ERR  SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                ; a sample program for LOCAL
0002                chip    1c864008
0003                b_ne   macro  val,dst
0004                local  skip
0005                be     val,skip
0006                bro    dst
0007                skip:
0008                endm
0009
0010                cseg
0011                b_ne   #0, over
0011+1              local  _L0000000L_
0011+2  C 0000 310003  be     #0,_L0000000L_
0011+3  C 0003 11FB01  bro    over
0011+4              _L0000000L_:
0012
0013                org    200h
0014                over:  b_ne   #0, under
0014+1              local  _L0000001L_
0014+2  C 0200 310002  be     #0,_L0000001L_
0014+3  C 0203 0101   bro    under
0014+4              _L0000001L_:
0015  C 0205 00       nop
0016  C 0206 00       under:  nop
0017                end

```

The identifier declared as LOCAL is replaced with different names.

*The format of the name generated is `_L#####L_` (where ##### is a serial number starting with 0)*

## IFDEF

Assemble if defined

### Syntax

IFDEF symbol

### Description

If symbol is defined, the statements after the IFDEF pseudoinstruction until the next ELSE or ENDIF are assembled.

### Example

```
page: 1 <ifdef.ASM>
ERR SEQ. S LOC. OBJ. SOURCE STATEMENTS
0001 ; a sample program for IFDEF
0002 chip 1c864024
0003 00000001 abc equ 1
0004 dseg
0005 D 0000 count: ds 1
0006
0007 cseg inblock
0008 C 0000 230010 mov #10h, acc
0009 ifdef abc
0010 C 0003 8302 add b
0011 C 0005 1200' st count
0012 else
0013 inc acc
0014 endif
0015 C 0007 A303 sub c
0016 ifdef efg
0017 add count
0018 endif
0019 end
```

Symbol efg is undefined  
so this section is not assembled.

Symbol abc is defined,  
so this section is assembled.

## IFNDEF

Assemble if undefined

### Syntax

IFNDEF symbol

## Description

If symbol is undefined, the statements after the IFNDEF pseudoinstruction until the next ELSE or ENDIF are assembled.

## Example

```

page:      1 <ifdef.ASM>
ERR SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                ; a sample program for IFDEF
0002                chip    lc864024
0003          00000001 abc    equ    1
0004                dseg
0005      D 0000          count: ds    1
0006
0007                cseg    inblock
0008      C 0000 230010      mov    #10h, acc
0009                ifdef   abc
0010      C 0003 8302                add    b
0011      C 0005 1200'              st     count
0012                else
0013                inc     acc
0014                endif
0015      C 0007 A303          sub    c
0016                ifdef   efg
0017                add    count
0018                endif
0019                end

```

Symbol efg is undefined  
so this section is not assembled.

Symbol abc is defined,  
so this section is assembled.

## IFB

### Assemble if operand empty

## Syntax

**IFB**    <argument>

## Description

If argument is empty, the statements after the IFB pseudoinstruction until the next ELSE or ENDIF are assembled. "Empty" means that there are no characters at all (even spaces or tabs) between the angle brackets in which argument must be enclosed.

## Example

```
page: 1 <ifb.ASM>
ERR SEQ. S LOC. OBJ. SOURCE STATEMENTS
0001 ; a sample program for IFB
0002 chip lc864016
0003 tifb macro arg
0004 ifb <<arg>> ← The angle brackets must
0005 inc a ← be double, because one
0006 else layer is removed in each
0007 inc b macro expansion.
0008 endif
0009 endm
0010
0011 tifb xxx
0011+1 ifb <xxx>
0011+2 inc a
0011+3 else
0011+4 C 0000 6302 inc b ←
0011+5 endif
0012 tifb
0012+1 ifb <>
0012+2 C 0002 6300 inc a ←
0012+3 else
0012+4 inc b
0012+5 endif This is assembled because the
0013 end argument to ifb is empty.

This is assembled because the
argument to IFB is nonempty.
```

### IFNB

#### Assemble if operand nonempty

#### Syntax

**IFNB** <argument>

#### Description

If argument is nonempty, the statements after the IFNB pseudoinstruction until the next ELSE or ENDIF are assembled. "Empty" means that there are no characters at all (even spaces or tabs) between the angle brackets in which argument must be enclosed.

## Example

```

page:      1 <ifnb.ASM>
ERR SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                ; a sample program for IFNB
0002                chip    lc864016
0003                tifb   macro  arg
0004                ifnb   <<arg>> ← The angle brackets must
0005                inc    a      ← be double, because one
0006                else   ← layer is removed in each
0007                inc    b      ← macro expansion.
0008                endif
0009                endm
0010
0011                tifb   xxx
0011+1             ifnb   <xxx>
0011+2 C 0000 6300    inc    a ←
0011+3             else
0011+4             inc    b
0011+5             endif
0012                tifb
0012+1             ifnb   <>
0012+2             inc    a
0012+3             else
0012+4 C 0002 6302    inc    b ←
0012+5             endif
0013                end

```

This is assembled because  
the argument to IFB is empty.

This is assembled because  
the argument to IFB is nonempty.

### IFE

#### Assemble if zero

#### Syntax

**IFE**    **expression**

#### Description

If the value of expression is zero, the statements after the IFE pseudoinstruction until the next ELSE or ENDIF are assembled.

## Example

```
page:      1 <ife.ASM>
ERR SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                ; a sample program for IFE
0002                chip  lc866032
0003                cseg
0004          00000003 aa  set    3
0005                ife    aa-2
0006                inc    70h
0007                else
0008 C 0000 7270          dec    70h ←
0009                endif
0010          00000002 aa  set    aa-1
0011                ife    aa-2
0012 C 0002 6270          inc    70h ←
0013                else
0014                dec    70h
0015                endif
0016                end
```

The expression value is zero,  
so this section is assembled.

The expression value is zero,  
so this section is assembled.

## IFNE

Assemble if nonzero

### Syntax

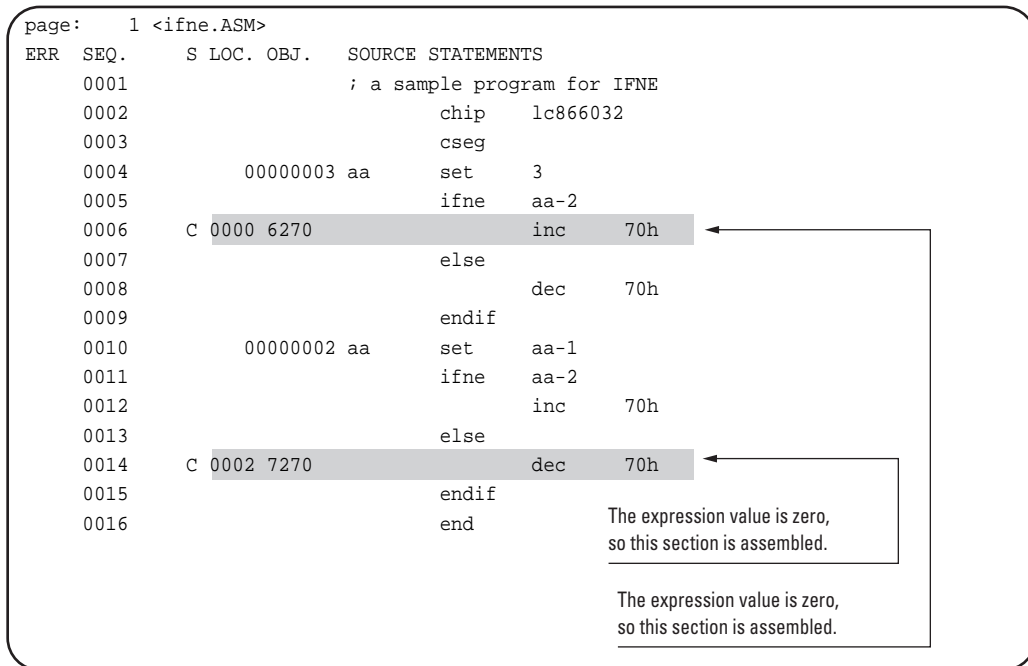
**IFNE** expression

### Description

If the value of expression is nonzero, the statements after the IFNE pseudoinstruction until the next ELSE or ENDIF are assembled.

## Example

```
page:      1 <ifne.ASM>
ERR SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                ; a sample program for IFNE
0002                chip   lc866032
0003                cseg
0004          00000003 aa   set    3
0005                ifne   aa-2
0006 C 0000 6270          inc   70h
0007                else
0008                dec    70h
0009                endif
0010          00000002 aa   set    aa-1
0011                ifne   aa-2
0012                inc    70h
0013                else
0014 C 0002 7270          dec    70h
0015                endif
0016                end
```



The expression value is zero,  
so this section is assembled.

The expression value is zero,  
so this section is assembled.

## IFIDN

Assemble if identical

### Syntax

**IFIDN** <string1>, <string2>

### Description

If the two strings string1 and string2 are identical, the statements after the IFIDN pseudoinstruction until the next ELSE or ENDIF are assembled. The strings must be enclosed in angle brackets, and within them, comparison is carried out with spaces and tabs considered significant.

## Example

```
page:      1 <ifidn.ASM>
ERR SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                ; a sample program for IFIDN
0002                chip    lc866032
0003                cseg
0004                tifidn  macro  arg1,arg2
0005                ifidn  <<arg1>>,<<arg2>>
0006                inc    a
0007                else
0008                dec    a
0009                endif
0010                endm
0011
0012                tifidn  same, same
0012+1             ifidn  <same>,<same>
0012+2 C 0000 6300    inc    a
0012+3             else
0012+4             dec    a
0012+5             endif
0013                tifidn  same, not_same
0013+1             ifidn  <same>,<not_same>
0013+2             inc    a
0013+3             else
0013+4 C 0002 7300    dec    a
0013+5             endif
0014                end
```

The angle brackets must be double, because one layer is removed in each macro expansion.

The strings are different, so this section is assembled.

The strings are the same, so this section is assembled.

## IFDIF

### Assemble if different

### Syntax

**IFDIF** <string1>, <string2>

### Description

If the two strings string1 and string2 are different, the statements after the IFDIF pseudoinstruction until the next ELSE or ENDIF are assembled. The strings must be enclosed in angle brackets, and within them, comparison is carried out with spaces and tabs considered significant.



## Example

```

page:      1 <ifdif.ASM>
ERR SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                ; a sample program for IFDIF
0002                chip   lc866032
0003                cseg
0004                tifidn macro  arg1,arg2
0005                ifdif  <<arg1>>,<<arg2>>
0006                inc    a
0007                else
0008                dec    a
0009                endif
0010                endm
0011
0012                tifidn same, same
0012+1             ifdif  <same>,<same>
0012+2             inc    a
0012+3             else
0012+4 C 0000 7300    dec    a
0012+5             endif
0013                tifidn same, not_same
0013+1             ifdif  <same>,<not_same>
0013+2 C 0002 6300    inc    a
0013+3             else
0013+4             dec    a
0013+5             endif
0014                end

```

The angle brackets must be double, because one layer is removed in each macro expansion.

the strings are different so this section is assembled.

The strings are the same so this section is assembled.

## ELSE

Else case of conditional assembly

## Syntax

## ELSE

## Description

The statements after the ELSE pseudoinstruction until the next ENDIF are assembled when the test condition of the preceding IF pseudoinstruction fails to hold.

**Reference:** See under "IFDEF - Assemble if defined" in this section.

**ENDIF**

**End conditional assembly**

**Syntax**

**ENDIF**

**Description**

Marks the end of a conditional assembly.

---

**Reference:** See under "IFDEF - Assemble if defined" in this section.

---

**.PRINTX**

**Display message during assembly**

**Syntax**

**.PRINTX"string"**

**Description**

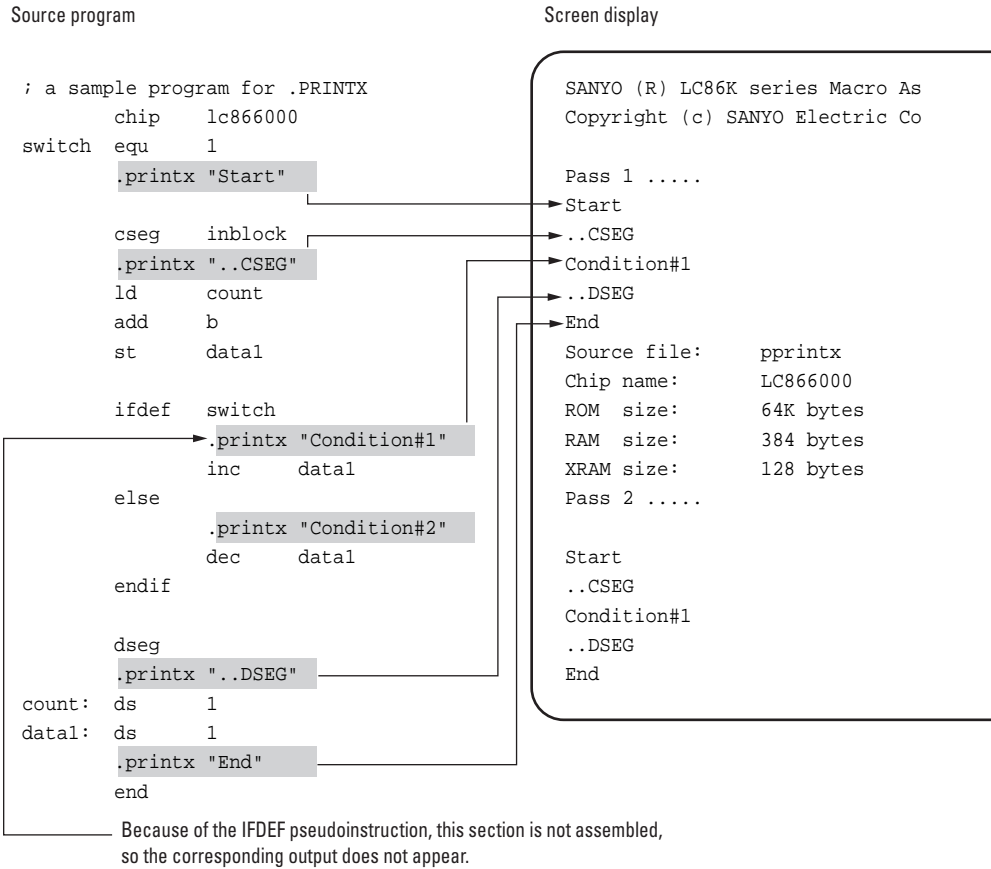
The .PRINTX pseudoinstruction displays the character string constant string during assembly.

---

**Reference:** For details of character string constants, see Section 20.7, "Character String Constants."

---

**Example**



**.LIST**

**Resume listing**

**Syntax**

**.LIST**

**Description**

The .LIST pseudoinstruction resumes listing output, when it has been suppressed with the .XLIST pseudoinstruction.

## Example

```
    ; a sample program for LIST
    chip    lc866200
    cseg    inblock
    mov     #00, count
    ld     count
    add     #10h
    st     b

    .xlist
    abc     equ    10h
           dseg
    count:  ds    4
    .list

    cseg    inblock
    ld     b
    sub     #abc
    st     count
    end

page:    1 <plist.ASM>
ERR SEQ.  S LOC. OBJ.  SOURCE STATEMENTS
0001                ; a sample program for LIST
0002                chip    lc866200
0003                cseg    inblock
0004    C 0000 220000'  mov     #00, count
0005    C 0003 0200'   ld     count
0006    C 0005 8110   add     #10h
0007    C 0007 1302   st     b
0008
0014                .list
0015                cseg    inblock
0016    C 0000 0302   ld     b
0017    C 0002 A110'  sub     #abc
0018    C 0004 1200'  st     count
0019                end
```

From the .XLIST line onwards, output to the listing file is suppressed. However, line numbers are still counted, so there is no loss of synch.

From the .LIST line onwards, output to the listings file is resumed.

### **.XLIST**

#### **Suppress listing**

#### **Syntax**

### **.XLIST**

#### **Description**

The .XLIST pseudoinstruction suppresses output to the listing file.

---

**Reference:** See under ".LIST - Resume listing," in this section.

---

**.MACRO**

**List macro expansions**

**Syntax**

**.MACRO**

**Description**

The .MACRO pseudoinstruction causes the expanded body of macro calls to be output to the listing file.

**Example**

```

; a sample program for .MACRO
    chip    lc866200
t.mac macro
    inc    a
    inc    b
    endm
cseg  inblock
    t.mac
    .xmacro
    t.mac
    .macro
    t.mac
    end
page: 1 <pmacro.ASM>
ERR SEQ.  S LOC. OBJ.  SOURCE STATEMENTS
0001                ; a sample program for .MACRO
0002                chip    lc866200
0003                t.mac macro
0004                    inc    a
0005                    inc    b
0006                    endm
0007
0008                cseg  inblock
0009                t.mac
0009+1 C 0000 6300    inc    a
0009+2 C 0002 6302    inc    b
0010                .xmacro
0011                t.mac
0012                .macro
0013                t.mac
0013+1 C 0008 6300    inc    a
0013+2 C 000A 6302    inc    b
0014                end

```

The .XMACRO pseudoinstruction ends the output of expanded macro calls to the listing. This means that the generated statements and code both disappear.

The .XMACRO pseudoinstruction resumes causes the listing of expanded macro calls.

**.XMACRO**

**End macro expansion listing**

**Syntax**

**.XMACRO**

**Description**

The .XMACRO pseudoinstruction ends the output of expanded macro calls to the listing.

---

**Reference:** For an example, see under the previous item, ".MACRO - List macro expansions."

---

**.IF**

**List skipped statements in conditional assembly**

**Syntax**

**.IF**

**Description**

The .IF pseudoinstruction causes source program statements skipped in a conditional assembly to be output to the listing file.

**Example**

```

; a sample program for .IF
chip    lc866200
t.if    macro    arg1
        ifb     <<arg1>>
            inc    a
        else
            inc    b
        endif
    endm
cseg    inblock
t.if
.xif
t.if    abc
.if
t.if    def
end

```

The .XIF pseudoinstruction stops source program statements skipped in a conditional assembly from being output to the listing file, the part of a conditional assembly section which is assembled appears in the listing regardless of this pseudoinstruction.

The .IF pseudoinstruction causes even the source program statements skipped in a conditional assembly to be output to the listing file.

```

page:    1 <pif.ASM>
ERR  SEQ.    S LOC. OBJ.  SOURCE STATEMENTS
0001                                ; a sample program for .IF
0002                                chip    lc866200
0003
0004                                t.if    macro    arg1
0005                                ifb     <<arg1>>
0006                                inc     a
0007                                else
0008                                inc     b
0009                                endif
0010                                endm
0011                                cseg    inblock
0012                                t.if
0012+1                                ifb     <>
0012+2                                C 0000 6300    inc     a
0012+3                                else
0012+4                                inc     b
0012+5                                endif
0013                                .xif
0014                                t.if    abc
0014+1                                ifb     <abc>
0014+3                                C 0002 6302    else
0014+4                                inc     b
0014+5                                endif
0015                                .if
0016                                t.if    def
0016+1                                ifb     <def>
0016+2                                inc     a
0016+3                                else
0016+4                                C 0004 6302    inc     b
0016+5                                endif
0017                                end

```

**.XIF**

**End listing of skipped statements**

**Syntax**

**.XIF**

**Description**

The .XIF pseudoinstruction stops source program statements skipped in a conditional assembly from being output to the listing file.

---

**Reference:** For an example, see under the previous item, ".IF - List skipped statements in conditional assembly."

---

**INCLUDE**

**Include file**

**Syntax**

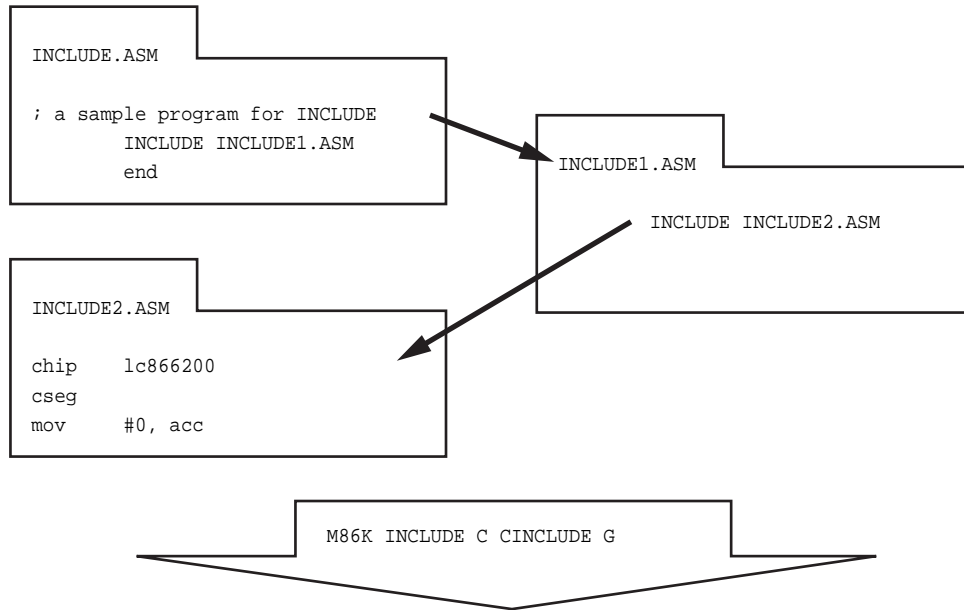
**INCLUDE filename**

**Description**

The INCLUDE pseudoinstruction causes the source file specified by filename to be read into the current point in the source program and assembled. The specification of filename must include the extension. The INCLUDE pseudoinstruction can be nested to a maximum depth of nine. Note that if an END pseudoinstruction occurs in the included file, this terminates the assembly.



**Example**



```

page:      1 <include.ASM>
ERR  SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
    0001                ; a sample program for INCLUDE
    0002                INCLUDE INCLUDE1.ASM
1/0001                INCLUDE INCLUDE2.ASM
2/0001                chip    lc866200
2/0002                cseg
2/0003                C 0000 230000    mov     #0, acc
    0003
    ↑ Indicates the nesting depth of includes.
    
```

**TITLE**

**Set listing title**

**Syntax**

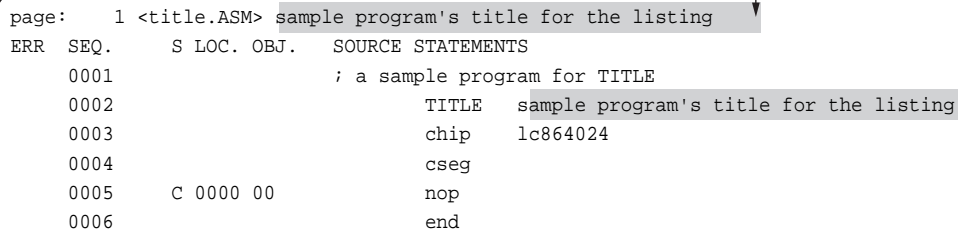
**TITLE string**

**Description**

The TITLE pseudoinstruction specifies string as the title for the listing file. Unlike a character string constant, string is not enclosed in quotation marks. It is also not possible to include codes with the backslash (\) symbol.

## Example

This string appears on all pages of the listing.



The screenshot shows an assembly listing with a callout box. The callout box is a rounded rectangle with a black border, containing the following text:

```
page: 1 <title.ASM> sample program's title for the listing
ERR SEQ. S LOC. OBJ. SOURCE STATEMENTS
0001 ; a sample program for TITLE
0002 TITLE sample program's title for the listing
0003 chip lc864024
0004 cseg
0005 C 0000 00 nop
0006 end
```

An arrow points from the text 'This string appears on all pages of the listing.' to the string 'sample program's title for the listing' in the listing, which is highlighted in grey in the original image.

## PAGE

### New page

### Syntax

## PAGE

### Description

The PAGE pseudoinstruction forces a new page in the listing file. The page break appears immediately after this pseudoinstruction.

### Example

## Source file

```
; a sample program for PAGE
  chip    lc866032
  page
  cseg
  page
  nop
  page
  end
```

## Listing file

```
page:    1 <page.ASM>
ERR SEQ.  S LOC. OBJ.  SOURCE STATEMENTS
0001                                ; a sample program for PAGE
0002                                chip    lc866032
```

```
page:    2 <page.ASM>
ERR SEQ.  S LOC. OBJ.  SOURCE STATEMENTS
0003                                page
0004                                cseg
```

```
page:    3 <page.ASM>
ERR SEQ.  S LOC. OBJ.  SOURCE STATEMENTS
0005                                page
0006    C 0000 00      nop
```

```
page:    4 <page.ASM>
ERR SEQ.  S LOC. OBJ.  SOURCE STATEMENTS
0007                                page
0008                                end
```

### **CHIP**

**Specify chip for assembly**

#### **Syntax**

**CHIP** chipname

#### **Description**

The CHIP pseudoinstruction informs the assembler of the chip for which assembly is to be carried out. According to the value of chipname, the assembly changes the reserved words, and carries out a memory size check. This pseudoinstruction must appear at the beginning of the source file, before any other instructions or pseudoinstructions. If this pseudoinstruction is not found, the environment variable CHIPNAME is referenced. If the chip name specified by this pseudoinstruction is different from the chip specified by the CHIPNAME environment variable, a warning level error is issued.

---

**Note:** For developing Visual Memory applications, the chip name must be set to LC868700.

---

### **COMMENT**

**Add comment to object file**

#### **Syntax**

**COMMENT** comment\_string

#### **Description**

The COMMENT pseudoinstruction adds a comment directly into the assembled object code. Unlike a character string constant, comment\_string is not enclosed in quotation marks. It is also not possible to include codes with the backslash (\) symbol. The comment is stored from byte 680 of the object file. A maximum of 255 characters can be used for the comment.

**Example**

Source file

```

; a sample program for COMMENT
  chip    lc866024
  comment This is a comment string embedded into OBJ file
  cseg
  nop
  end
    
```

Dump of object file (part only)

		Character count (1 byte)										
00000260	00 00 00 00 00 00 00 00 00-00 00	00	00	00	00	00	00	00	00	00	00	.....
00000270	00 00 00 00 00 60 00 00-80 01	00	00	80	00	00	00	00	00	00	00	.....
00000280	C6 92 40 2B 4D 38 36 4B-20 20	20	20	20	63	6F	6D	6D	6D	6D	6D	**+M86K comm
00000290	65 6E 74 2E 41 53 4D 20-63 6F	6D	6D	65	6E	74	20	ent.ASM comment				
000002A0	4C 43 38 36 36 30 32 34-30 54	68	69	73	20	69	73	LC8660240This is				
000002B0	20 61 20 63 6F 6D 6D 65-6E 74	20	73	74	72	69	6E	a comment strin				
000002C0	67 20 65 6D 62 65 64 64-65 64	20	69	6E	74	6F	20	g embedded into				
000002D0	4F 42 4A 20 66 69 6C 65-00 00	01	01	00	01	00	05	OBJ file.....				
000002E0	00 01 00 00 00 00 00 00-00 00	E0	00	00	00	00	C4	.....*				
000002F0	00 00 00 00 C4 00 00 00-00 24	00	00	01	00	04	01	....*....\$......				
00000300	00 00 00 24							...\$				

**WIDTH**

Specify columns in listing file

**Syntax**

**WIDTH** number

**Description**

The WIDTH pseudoinstruction specifies the number of character columns in the listing file, that is, the number of characters in each line. The parameter number may be any value from 72 to 132 inclusive, but the recommended minimum is the number of columns of the source file plus 28. Although this pseudoinstruction can appear any number of times in a single source file, normally it is specified once only at the beginning of the file. If this pseudoinstruction is not found, the default listing file has 132 columns.

## Example

WIDTH is evaluated on pass 1 and pass 2, but the listing output occurs in pass 2 only, so the last value found in pass 1 is used here, causing the lines to be folded at 78 characters

```
1 2 3 4 5 6 7 8
1234567890123456789012345678901234567890123456789012345678901234567890
page: 1 <width.ASM>
ERR SEQ. S LOC. OBJ. SOURCE STATEMENTS
0001 ; a sample program for WIDTH
0002 chip lc866200
0003 cseg ; this is a long line to indicat
0003 e WIDTH's effect
0004 WIDTH 72
0005 C 0000 00 nop ; this is also a long line
0005 to indicate WIDTH's effect
0006 WIDTH 78
0007 end
```

Carriage return lines and linefeed is inserted at character position 72, folding the lines here.

## BANK

### Specify RAM bank

### Syntax

### BANK expression

### Description

The BANK pseudoinstruction supplies the bank number for symbols defined by DS pseudoinstructions for RAM after a DSEG pseudoinstruction.

## Example

```

page:      1 <bank.ASM>
ERR  SEQ.   S LOC. OBJ.   SOURCE STATEMENTS
0001                               ; a sample program for BANK
0002                               chip    lc866032
0003                               cseg   inblock
0004
0005      C 0000 220000'      mov    #0,data1
0006
0007      C 0003 6200'       inc    data1
0008      C 0005 0200'       ld     data1
0009      C 0007 1201'       st     data2
0010
0011      C 0009 6200'       inc    dataa
0012      C 000B 0200'       ld     dataa
0013      C 000D 1202'       st     datac
0014
0015                               dseg
0016                               bank    0
0017      D 0000      data1: ds     1
0018      D 0001      data2: ds     1
0019      D 0002      data3: ds     1
0020
0021                               bank    1
0022      D 0000      dataa: ds     1
0023      D 0001      datab: ds     1
0024      D 0002      datac: ds     1
0025
0026                               end

```

← These symbols are assigned to bank 1.

← These symbols are assigned to bank 0.

### CHANGE

#### Jump between flash memory and ROM

#### Syntax

#### CHANGE symbol

#### Description

For the LC86800 series, this is a special jump instruction for switching between code in flash memory and code in ROM (system BIOS). The operand symbol must have been declared with the pseudoinstruction OTHER\_SIDE\_SYMBOL. Note that this pseudoinstruction is special to the LC86800 series, and in other cases an error results.

---

**Note:** For Visual Memory, use this instruction to call an operating system function.

---

**Reference:** For details of operating system function calls, refer to the "System BIOS" section of the Visual Memory Hardware Manual.

---

## Example

```
page:      1 <change.ASM>
ERR SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                      ; a sample program for CHANGE
0002                      chip    lc868032
0003                      other_side_symbol    far_away
0004
0005                      cseg
0006      C 0000 B80D21'    change far_away
0006      C 0003 0000'
```

## RADIX

### Specify default radix

### Syntax

**RADIX** expression

### Description

The RADIX pseudoinstruction specifies the radix, or base, of a numeric constant with no explicit radix indication. The value of expression must be a constant value from the set 2, 8, 10, and 16. This specification takes effect from this statement until a subsequent RADIX pseudoinstruction. If this pseudoinstruction is not present, the default radix is 10.

### Example

```
Xxx      SET    10    _   interpreted by default as 10 decimal.
          RADIX  16
Xxx      SET    10    _   interpreted as 16 decimal, because of the radix value 16.
          RADIX  2
Xxx      SET    10    _   interpreted as 2 decimal, because of the radix value 2.
```

## JMPO

### Optimized JMP instruction

### Syntax

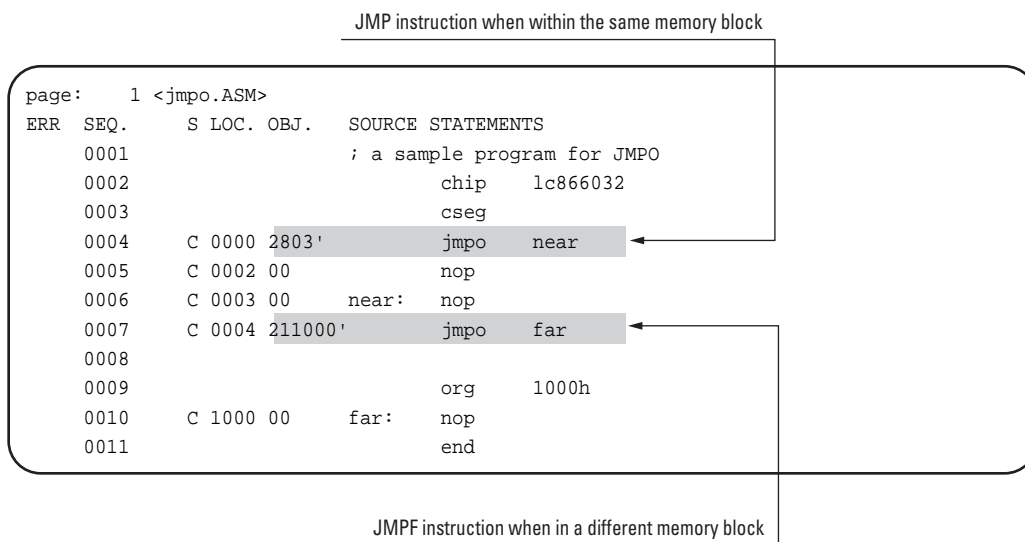
**JMPO** expression



## Description

The JMPO pseudoinstruction compares expression with the current location, and if this is a jump within the same block (only the bottom 12 bits of the addresses are different) generates a JMP instruction. Otherwise, that is, if the address is in a different block, or if the address cannot be determined because for example it is an external symbol, then this generates a JMPF instruction.

## Example



## BRO

### Optimized BR instruction

### Syntax

**BRO** expression

### Description

BRO pseudoinstruction compares expression with the current location, and if the branch address is within the range -128 to +127 generates a BR instruction; when outside the range -128 to +127 generates a BRF instruction.

## Example

Generates a BR instruction because destination is within the range -128 to +127.

```
page:      1 <bro.ASM>
ERR SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                ; a sample program for BRO
0002                chip   1c866032
0003                cseg
0004      C 0000 0101    bro   near ←
0005      C 0002 00          nop
0006      C 0003 00    near:  nop
0007      C 0004 11FA00   bro   far ←
0008
0009                org    100h
0010      C 0100 00    far:   nop
0011                end
```

Generates a BRF instruction because destination is outside the range -128 to +127.

## CALLO

### Optimized CALL instruction

### Syntax

**CALLO** expression

### Description

The CALLO pseudoinstruction compares expression with the current location, and if this is a call within the same block (only the bottom 12 bits of the addresses are different) generates a CALL instruction. Otherwise, that is, if the address is in a different block, or if the address cannot be determined because for example it is an external symbol, then this generates a CALLF instruction.

## Example

CALL instruction when within the same memory block.

```

page:      1 <CALLo.ASM>
ERR SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                               ; a sample program for CALLO
0002                               chip    1c866032
0003                               cseg
0004      C 0000 0805'          CALLo  near ←
0005      C 0002 201000'        CALLo  far  ←
0006
0007      C 0005 00          near:  nop
0008      C 0006 A0          ret
0009
0010                               org    1000h
0011      C 1000 00          far:   nop
    
```

CALLF instruction when in a different memory block.

## BZO

### BZ instruction guaranteeing no address error

### Syntax

**BZO** expression

### Description

The BZO macro generates code equivalent to the BZ instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BZO macro uses a BNZ instruction, which is the logical inverse of the BZ instruction, and a BRO instruction. Enter the branch destination for expression.

### Code generation macro

```

; *** Branch near relative address if accumulator is zero ***
bzo          macro                r8
              local                _next_
              bnz                   _next_
              bro                    r8
_next_:
              endm
    
```

### BNZO

**BNZ instruction guaranteeing no address error**

#### Syntax

**BNZO** expression

#### Description

The BNZO macro generates code equivalent to the BNZ instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BNZO macro uses a BZ instruction, which is the logical inverse of the BNZ instruction, and a BRO instruction. Enter the branch destination for expression.

#### Code generation macro

```
; *** Branch near relative address if accumulator is not zero ***
bnzo          macro          r8
              local         _next_
              bz            _next_
              bro           r8
_next_:
              endm
```

### BPO

**BP instruction guaranteeing no address error**

#### Syntax

**BPO** expression

#### Description

The BPO macro generates code equivalent to the BP instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BPO macro uses a BP instruction, and BR and BRO instructions. Enter the branch destination for expression.

**Code generation macro**

```
; *** Branch near relative address if direct bit is one ***
bpo          macro          d9,b3,r8
              local        _next_
              local        _true_
              bp           d9,b3,_true_
              br           _next_
_true_:      bro           r8
_next_:
              endm
```

**BPCO****BPC instruction guaranteeing no address error****Syntax****BPCO expression****Description**

The BPCO macro generates code equivalent to the BPC instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BPCO macro uses a BPC instruction, and BR and BRO instructions. Enter the branch destination for expression.

**Code generation macro**

```
; *** Branch near relative address if direct bit is one,
; and clear ***
bpc          macro          d9,b3,r8
              local        _next_
              local        _true_
              bpc          d9,b3,_true_
              br           _next_
_true_:      bro           r8
_next_:
              endm
```

### BNO

**BN instruction guaranteeing no address error**

#### Syntax

**BNO** expression

#### Description

The BNO macro generates code equivalent to the BN instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BNO macro uses a BN instruction, and BR and BRO instructions. Enter the branch destination for expression.

#### Code generation macro

```
; *** Branch near relative address if direct bit is zero ***
bno          macro          d9,b3,r8
              local        _next_
              local        _true_
              bn            d9,b3,_true_
              br            _next_
_true_:      bro            r8
_next_:
              endm
```

### DBNZO

**DBNZ instruction guaranteeing no address error**

#### Syntax

**DBNZO** expression

#### Description

The DBNZO macro generates code equivalent to the DBNZ instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The DBNZO macro uses a DBNZ instruction, and BR and BRO instructions. The function of expression is the same as in the DBNZ instruction.

**Code generation macro**

```
; *** Decrement direct byte and branch near relative address
;           if direct byte is not zero ***
dbnzo      macro                                d9,r8
            local                                _next_
            local                                _true_
            dbnz                                 d9,_true_
            br                                   _next_
_true_:    bro                                   r8
_next_:
            endm
```

**BEO****BE instruction guaranteeing no address error****Syntax****BEO** expression**Description**

The BEO macro generates code equivalent to the BE instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BEO macro uses a BNE instruction and BRO instruction. The function of expression is the same as in the BE instruction.

**Code generation macro**

```
; *** Compare immediate data or accumulator and branch
;           near relative address if equal ***
beo        macro                                arg0,arg1,arg2
            local                                _next_
            local                                _txen_
            ifb                                  <<arg2>>
            bne                                  arg0,_next_
            bro                                  arg1
_next_:
            else
            bne                                  arg0,arg1,_txen_
            bro                                  arg2
_txen_:
            endif
            endm
```

### BNEO

**BNE instruction guaranteeing no address error**

### Syntax

**BNEO expression**

### Description

The BNEO macro generates code equivalent to the BNE instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BNEO macro uses a BE instruction and BRO instruction. The function of expression is the same as in the BNE instruction.

### Code generation macro

```
; *** Compare immediate data or accumulator and branch
;          near relative address if equal ***
bneo      macro          arg0,arg1,arg2
          local         _next_
          local         _txen_
          ifb           <<arg2>>
          be            arg0,_next_
          bro           arg1

_next_:

          else
          be            arg0,arg1,_txen_
          bro           arg2

_txen_:

          endif
          endm
```



# ***LC86K Instruction Summary***

This chapter describes general features of flag handling and addressing, before the complete listing of the instruction set.

## **Instruction Summary**

### **Arithmetic Instructions**

The arithmetic instructions operate principally on the accumulator, and carry out the four basic operations, incrementing, and decrementing. The carry, auxiliary carry, and overflow flags are set according to the results of arithmetic operations, as follows.

**Table 2.38** *CY (carry flag)*

Arithmetic operation	Operation result	CY
Add instructions	When there is a carry from bit 7 (MSB)	1
	When there is no carry from bit 7 (MSB)	0
Subtraction and comparison instructions	When a borrow from bit 7 (MSB) is required	1
	When no borrow from bit 7 (MSB) is required	0
Multiplication and division instructions	-	0

**Table 2.39 AC (auxiliary carry flag)**

Arithmetic operation	Operation result	AC
Add instructions	When there is a carry from bit 3	1
	When there is no carry from bit 3	0
Subtraction instructions	When a borrow from bit 3 is required	1
	When no borrow from bit 3 is required	0

**Table 2.40 OV (overflow flag)**

Arithmetic operation	Operation result	OV
Add and subtract instructions	When there is a carry from bit 7 but not from bit 6	1
	When there is a carry from bit 6 but not from bit 7	1
	When an overflow error occurs in a signed variable addition instruction	1
	All other cases	0
Multiplication instructions	When the product is 256 or more	1
	When the product is 255 or less	0
Division instructions	When the divisor is zero	1
	When the divisor is nonzero	0

## Logical Instructions

Logical instructions carry out logical rotates. The RORC and ROLC instructions also affect the carry flag.

## Data Transfer Instructions

The data transfer instructions read, write, back up, and exchange data to and from RAM and special function registers (SFR).

## Jump Instruction

A jump instruction is an unconditional transfer to a new instruction.

## Conditional Branch Instructions

Conditional branch instructions determine the value of a specified condition as true or false, and transfer to the specified destination if true. If false, there is no transfer, and control passes to the next instruction.

The BE and BNE instructions branch on the basis of a comparison of two 8-bit data values, and the carry flag is set or cleared according to the result, as follows.

Operands	Carry flag (CY)			
	#i8, r8	d9, r8	@Rj, #i8, r8	
Magnitude relation	#i8 > (ACC)	(d9) > (ACC)	#i8 > ((Rj))	1
	#i8 = (ACC)	(d9) = (ACC)	#i8 = ((Rj))	0
	(d9) < (ACC)	(d9) < (ACC)	#i8 < ((Rj))	0

## Subroutine Instruction

The subroutine instruction performs an unconditional branch to another instruction. An address is stored on the stack in order that, after the branch, a return instruction (RET or RETI) can return to the instruction following the CALL instruction. The stack is in RAM, and is pointed to by the stack pointer (SP). Enough RAM must be reserved for the stack to allow for the nesting level of subroutine calls.

---

**Note:** The Visual Memory stack is held in bank 0 of RAM. When an application is started, the system BIOS sets it to 7FH. When a value is pushed onto the stack, the stack pointer is incremented before storing the data, so the actual values are stored from address 80H. The stack consumes addresses upwards from 7FH to 0FFH.

The internal clock function also needs 20 bytes on the stack.

---

## Bit Manipulation Instructions

The bit manipulation instructions operate on individual bits of specified RAM or special function registers (SFR).

## Other Instructions

The NOP instruction has no effect other than to consume one clock cycle.

## Macro Instruction

This is a dedicated standard macro instruction. It switches between the execution of the system BIOS in ROM and the application program in flash memory.

## Addressing

There are a number of different methods of addressing for flash memory, RAM, and special function registers (SFRs).

### Program Memory Addressing

The program ROM address of the destination of a jump, branch, or subroutine instruction is shown by the instruction code. In this case the address is shown by one of the following addressing methods.

#### **r8 (8-bit relative addressing)**

The transfer is to an address in the range -128 to +127 from the start address of the currently executed instruction. This is shown by a signed 8-bit value.

[80H to 7FH: -128 to +127]

#### **r16 (16-bit relative addressing)**

This allows a transfer anywhere within the 64K-byte flash memory address space. It is shown by an unsigned 16-bit value.

[0000H to FFFFH: +0 to +65535]

#### **a12 (12-bit absolute addressing)**

The top four bits PC15 to PC12 (the current page) of the address of the instruction after the current instruction (PC15 to PC00) are kept the same, and the remaining 12 bits (PC11 to PC00) are replaced by the address data (000H to FFFH). This allows a jump anywhere within the current page (PC15 to PC12).

---

**Caution:** Care is required, because if a JMP instruction or CALL instruction occurs at the final address within a page, the current page changes.

---

#### **a16 (16-bit absolute addressing)**

This allows a transfer anywhere within the 64K-byte flash memory address space.

The 16-bit value is used unchanged as the address.

[0000H to FFFFH: 0 to 65535]

#### **Table jumps**

If the destination address of a jump is on the stack, a RET instruction forces the address into the program counter (PC), thus achieving a jump.

In Example 1, the first line sets the stack pointer (SP) to 09H. Executing a RET instruction now causes a jump to the address whose upper byte is the value of byte 08H in RAM, and whose lower byte is the value of byte 07H in RAM; the jump address is set accordingly in lines 2 and 3.

Since the jump destination is PC = 0C13H, in line 2 the lower byte is set to 13H, and in line 3 the upper byte is set to 0CH. In the fourth line, when the RET instruction is executed, the stack pointer is set to 07H, and a jump to 0C13H occurs. However, in Example 1, since the stack pointer value must be known explicitly, normally a PUSH instruction is used as in Example 2.

**Example 1**

```
MOV          #09H,SP
MOV          #13H,07H
MOV          #0CH,08H
RET
```

**Example 2**

```
MOV          #13H,ACC
PUSH ACC
MOV          #0CH,ACC
PUSH ACC
RET
```

Example 3 carries out a 128-way branch to 00H to 7FH on the basis of the values of RAM address 70H.

In lines 1 and 2, the lower byte of the branch destination address is set, and in line 4 the upper byte of the address. The RET instruction in line 6 branches to the jump table in lines 7 and 8, thence to the branch destination.

This is referred to as a "table jump" and can be used to branch to a number of different addresses according to conditions.

**Example 3**

```
A0:          LD          070H
ROL
ADD #LOW(A1)
PUSH          ACC
MOV #HIGH(A1),ACC
PUSH          ACC
RET
;
ORG 0C00H
A1: JMP          B00          jump table
          -
          JMP          B7F
;
B00:          XXXXXX
```

## RAM and Special Function Register (SFR) Addressing

### d9 (direct addressing)

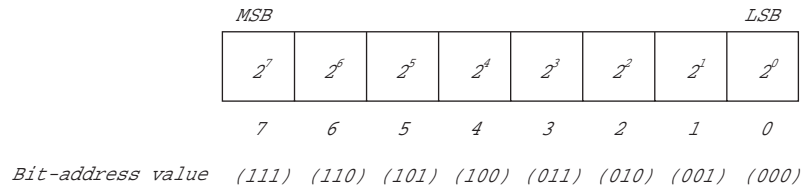
Addresses RAM or SFR directly with nine bits d8 to d0.

Addresses 000H to 0FFH . . . specify RAM.

Addresses 100H to 1FFH . . . specify an SFR.

### b3 (bit addressing)

In the bit manipulation instructions (SET1, CLR1, NOT1), and the BP, BPC, and BN instructions, 3-bit bit-address data is used in combination with d9 (direct addressing), to specify individual bits within RAM or an SFR.



### @Rj (indirect addressing)

For indirect addressing the destination RAM or SFR address is stored in a particular location in RAM, and the access made through specification of this address in RAM.

---

**Reference:** For more details of indirect addressing refer to the Visual Memory Hardware Manual.

---

The particular addresses in RAM are referred to as indirect address registers, and are indicated as @R0, @R1, @R2, and @R3. The indirect address registers are accessed using a 2-bit indirect addressing value (j1, j0), allowing a specification from @R0 to @R3.

A bank of four indirect address registers is assigned to the first 16 bytes (addresses 00H to 0FH) of each RAM bank. The RAM bank is selected with RAMBK0 (bit 1 of PSW). The indirect address register bank is selected with IRBK1 and 0 (bits 4 and 3 of PSW).

When an indirect addressing instruction is executed, for the indirect address register and the RAM address specified by the indirect address register, the RAM address used is in the RAM bank specified by IRBK1 and 0 and RAMBK0. On a reset, IRBK0, and 1 are both zero, and RAMBK0 is also set to zero, so the absolute addresses of @R0, @R1, @R2, and @R3 are respectively 00H, 01H, 02H, and 03H in RAM bank 0.

Indirect address registers . . . @R3 @R2 @R1 @R0

Indirect addressing values (j1, j0) . . . (11) (10) (01) (00)

Indirect addressing register map

Indirect address register	Function	Bank 0 (IRBK1 = 0) (IRBK0 = 0)	Bank 1 (IRBK1 = 0) (IRBK0 = 1)	Bank 2 (IRBK1 = 1) (IRBK0 = 0)	Bank 3 (IRBK1 = 1) (IRBK0 = 1)
@R0	RAM access	RAM 00H	RAM 04H	RAM 08H	RAM 0CH
@R1	RAM access	RAM 01H	RAM 05H	RAM 09H	RAM 0DH
@R2	SFR access	RAM 02H	RAM 06H	RAM 0AH	RAM 0EH
@R3	SFR access	RAM 03H	RAM 07H	RAM 0BH	RAM 0FH

**Examples of indirect addressing**

The following are examples of calculation using indirect address registers.

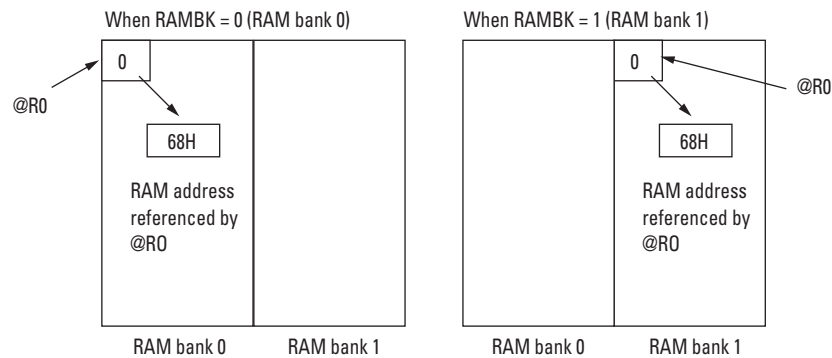
In Example 1, in the second line immediate data 68H is stored in RAM (address 00H). Using RAM (address 00H) as an indirect address register, RAM (address 68H) is accessed. For example, in line 3, the indirect address register (@R0) is specified to store immediate data 10H in RAM (address 68H).

In line 5, by specifying the indirect address register (@R0), the contents of RAM (address 68H) is added to the accumulator.

**Example 1**

```

MOV          #055H, ACC
MOV          #068H, 00H
MOV          #010H, @R0
ADD         #015H
ADD         @R0
    
```



The next example uses indirect addressing to access an SFR.

In Example 2, the first two lines clear bits 4 and 3 of the PSW, selecting RAM addresses 00H to 03H for the indirect address registers. In the fourth line immediate data 02H is stored in RAM address 02H. Then using RAM (address 02H) as an indirect address register accesses RAM (address 02H). For example, in line 5, immediate data 12H is stored by specifying the indirect address register (@R2) in an SFR (address 02H: B register). In line 6, the indirectly addressed B register is incremented.

### Example 2

```
CLR1          PSW, 4
CLR1          PSW, 3
MOV           #0ACH, ACC
MOV           #002H, 02H
MOV           #012H, @R2
INC           @R2
```

The next example uses bank-switching with the PSW, to indirectly address an SFR.

In Example 3, the first two lines set the PSW bank to 2, so that RAM addresses 08H to 0BH are used as indirect address registers. In line 4, immediate data 02H is stored in RAM address 0BH. Using RAM (address 0BH) as an indirect address register accesses RAM (address 02H). For example, in line 5 immediate data 12H is stored in the SFR (address 02H: B register) by specifying the indirect address register (@R2). In line 6 the indirectly addressed B register is incremented.

### Example 3

```
SET1          PSW, 4
CLR1          PSW, 3
MOV           #0ACH, ACC
MOV           #002H, 0BH
MOV           #012H, @R2
INC           @R2
```



# ***Instruction Set Reference***

The comprehensive LC86K instruction set includes some 70 instructions. Identified by some 45 operation codes these can be grouped into the following eight categories.

Arithmetic instructions	ADD, ADDC, SUB, SUBC, INC, DEC, MUL, DIV
Logical instructions	AND, OR, XOR, ROL, ROLC, ROR, RORC
Data transfer instructions	LD, ST, MOV, LDC, PUSH, POP, XCH
Jump instructions	JMP, JMPF, BR, BRF
Conditional branch instructions	BZ, BNZ, BP, BPC, BN, DBNZ, BE, BNE
Subroutine instructions	CALL, CALLF, CALLR, RET, RETI
Bit manipulation instructions	CLR1, SET1, NOT1
Miscellaneous instruction	NOP
Macro instruction	CHANGE

## Arithmetic Instructions

### ADD\_i8

#### ADD immediate data to accumulator

Instruction code	1 0 0 0 0 0 0 1 i7i6i5i4i3i2i1i0 (81H)
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) + #i8
Flags affected	CY, AC, OV
Interrupts enabled	Yes

#### Description

Adds the contents of the accumulator and immediate data (i7 to i0), and stores the result in the accumulator.

#### Example

		ACC	CY	AC	OV
MOV	#055H,ACC	55H	-	-	-
ADD	#013H	68H	0	0	0
ADD	#00AH	72H	0	1	0
ADD	#00FH	81H	0	1	1
ADD	#080H	01H	1	0	1

**ADD d9**

**ADD direct byte to accumulator**

Instruction code	1 0 0 0 0 0 1d8 d7d6d5d4d3d2d1d0 (82H to 83H)
Byte count	2
Cycles	1
Function	(ACC) _ (ACC) + (d9)
Flags affected	CY, AC, OV
Interrupts enabled	Yes

**Description**

Adds the contents of the accumulator and the contents of the RAM address or SFR specified by d8 to d0, and stores the result in the accumulator.

**Example 1**

		ACC	RAM	CY	AC	OV
			23H			
MOV	#055H,ACC	55H	-	-	-	-
MOV	#068H,023H	55H	68H	-	-	-
ADD	#00CH	61H	68H	0	1	0
ADD	023H	C9H	68H	0	0	1

**Example 2**

		ACC	B	CY	AC	OV
MOV	#070H,ACC	70H	-	-	-	-
MOV	#095H,B	70H	95H	-	-	-
ADD	#002H	72H	95H	0	0	0
ADD	B	07H	95H	1	0	0

## ADD @Rj

### ADD indirect byte to accumulator

Instruction code	1 0 0 0 0 1j1j0 (84H to 87H)
Byte count	1
Cycles	1
Function	$(ACC) \leftarrow (ACC) + ((Rj))$ j = 0, 1, 2, 3
Flags affected	CY, AC, OV
Interrupts enabled	Yes

### Description

Adds the contents of the accumulator and the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0, and stores the result in the accumulator.

### Example 1

		ACC	RAM	RAM	CY	AC	OV
	00H	68H					
MOV	#055H,ACC	55H	-	-	-	-	-
MOV	#068H,000H	55H	68H	-	-	-	-
MOV	#010H,@R0	55H	68H	10H	-	-	-
ADD	#015H	6AH	68H	10H	0	0	0
ADD	@R0	7AH	68H	10H	0	0	0

### Example 2

		ACC	RAM	TRL	CY	AC	OV
			02H				
MOV	#0AAH,ACC	AAH	-	-	-	-	-
MOV	#004H,002H	AAH	04H	-	-	-	-
MOV	#055H,@R2	AAH	04H	55H	-	-	-
ADD	#001H	ABH	04H	55H	0	0	0
ADD	@R2	00H	04H	55H	1	1	0

**ADDC\_i8****ADD immediate data and carry flag to accumulator**

Instruction code    1 0 0 1 0 0 0 1 i7i6i5i4i3i2i1i0 (91H)  
Byte count            2  
Cycles                1  
Function               $(ACC) \leftarrow (ACC) + (CY) + \#i8$   
Flags affected        CY, AC, OV  
Interrupts enabled    Yes

**Description**

Adds the contents of the accumulator and carry flag to the immediate data (i7 to i0), and stores the result in the accumulator.

**Example**

		ACC	CY	AC	OV
MOV	#055H,ACC	55H	-	-	-
ADD	#013H	68H	0	0	0
ADDC	#00AH	72H	0	1	0
ADDC	#00FH	81H	0	1	1
ADDC	#080H	01H	1	0	1
ADDC	#001H	03H	0	0	0

## ADDC d9

### ADD direct byte and carry flag to accumulator

Instruction code	1 0 0 1 0 0 1d8 d7d6d5d4d3d2d1d0 (92H to 93H)
Byte count	2
Cycles	1
Function	$(ACC) \leftarrow (ACC) + (CY) + (d9)$
Flags affected	CY, AC, OV
Interrupts enabled	Yes

### Description

Adds the contents of the accumulator and carry flag to the contents of the RAM address or SFR specified by d8 to d0, and stores the result in the accumulator.

### Example 1

		ACC	RAM	CY	AC	OV
			23H			
MOV	#055H, ACC	55H	-	-	-	-
MOV	#068H, 023H	55H	68H	-	-	-
ADD	#00CH	61H	68H	0	1	0
ADDC	023H	C9H	68H	0	0	1
SET1	PSW, 7	C9H	68H	1	0	1
ADDC	023H	32H	68H	1	1	0

### Example 2

		ACC	B	CY	AC	OV
MOV	#070H, ACC	70H	-	-	-	-
MOV	#095H, B	70H	95H	-	-	-
ADD	#002H	72H	95H	0	0	0
ADDC	B	07H	95H	1	0	0
ADDC	B	9DH	95H	0	0	0

**ADDC @Rj**

**ADD indirect byte and carry flag to accumulator**

Instruction code	1 0 0 1 0 1j1j0 (94H to 97H)
Byte count	1
Cycles	1
Function	$(ACC) \leftarrow (ACC) + (CY) + ((Rj))$ j = 0, 1, 2, 3
Flags affected	CY, AC, OV
Interrupts enabled	Yes

**Description**

Adds the contents of the accumulator and carry flag to the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0, and stores the result in the accumulator.

**Example 1**

		ACC	RAM	RAM	CY	AC	OV
			00H	68H			
MOV	#055H,ACC	55H	-	-	-	-	-
MOV	#068H,000H	55H	68H	-	-	-	-
MOV	#010H,@R0	55H	68H	10H	-	-	-
ADD	#015H	6AH	68H	10H	0	0	0
ADDC	@R0	7AH	68H	10H	0	0	0
SET1	PSW,7	7AH	68H	10H	1	0	0
ADDC	@R0	8BH	68H	10H	0	0	1

**Example 2**

		ACC	RAM	TRL	CY	AC	OV
			02H				
MOV	#0AAH,ACC	AAH	-	-	-	-	-
MOV	#004H,002H	AAH	04H	-	-	-	-
MOV	#055H,@R2	AAH	04H	55H	-	-	-
ADD	#001H	ABH	04H	55H	0	0	0
ADDC	@R2	00H	04H	55H	1	1	0
ADDC	@R2	56H	04H	55H	0	0	0

## SUB\_i8

### Subtract immediate data from accumulator

Instruction code            1 0 1 0 0 0 0 1 i7i6i5i4i3i2i1i0 (A1H)  
 Byte count                2  
 Cycles                    1  
 Function                  (ACC) ← (ACC) - #i8  
 Flags affected            CY, AC, OV  
 Interrupts enabled        Yes  
 Description

Subtracts immediate data (i7 to i0) from the contents of the accumulator, and stores the result in the accumulator.

### Example

		ACC	CY	AC	OV
MOV	#055H,ACC	55H	-	-	-
SUB	#013H	42H	0	0	0
SUB	#003H	3FH	0	1	0
SUB	#03FH	00H	0	0	0
SUB	#002H	FEH	1	1	0

## SUB\_d9

### Subtract direct byte from accumulator

Instruction code        1 0 1 0 0 0 1d8 d7d6d5d4d3d2d1d0 (A2H to A3H)  
 Byte count            2  
 Cycles                1  
 Function              (ACC) ← (ACC) - (d9)  
 Flags affected        CY, AC, OV  
 Interrupts enabled    Yes  
 Description

Subtracts the contents of the RAM address or SFR specified by d8 to d0 from the contents of the accumulator, and stores the result in the accumulator.

### Example 1

		ACC	RAM	CY	AC	OV
			23H			
MOV	#055H,ACC	55H	-	-	-	-
MOV	#068H,023H	55H	68H	-	-	-
SUB	#00CH	49H	68H	0	1	0
SUB	023H	E1H	68H	1	0	0

### Example 2

		ACC	RAM	CY	AC	OV
MOV	#080H,ACC	80H	-	-	-	-
MOV	#095H,B	80H	95H	-	-	-
SUB	#002H	7EH	95H	0	1	1
SUB	B	E9H	95H	1	0	1



**SUB @Rj**

**Subtract indirect byte from accumulator**

Instruction code	1 0 1 0 0 1j1j0 (A4H to A7H)
Byte count	1
Cycles	1
Function	(ACC) ← (ACC) - ((Rj)) j = 0, 1, 2, 3
Flags affected	CY, AC, OV
Interrupts enabled	Yes

**Description**

Subtracts the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0 from the contents of the accumulator, and stores the result in the accumulator.

**Example 1**

		ACC	RAM	RAM	CY	AC	OV
			00H	68H			
MOV	#055H,ACC	55H	-	-	-	-	-
MOV	#068H,00H	55H	68H	-	-	-	-
MOV	#010H,@R0	55H	68H	10H	-	-	-
SUB	#016H	3FH	68H	10H	0	1	0
SUB	@R0	2FH	68H	10H	0	0	0

**Example 2**

		ACC	RAM	TRL	CY	AC	OV
			02H				
MOV	#0AAH,ACC	AAH	-	-	-	-	-
MOV	#004H,002H	AAH	04H	-	-	-	-
MOV	#0AAH,@R2	AAH	04H	AAH	-	-	-
SUB	#001H	A9H	04H	AAH	0	0	0
SUB	@R2	FFH	04H	AAH	1	1	0

## SUBC\_i8

### Subtract immediate data and carry flag from accumulator

Instruction code    1 0 1 1 0 0 0 1 i7i6i5i4i3i2i1i0 (B1H)  
Byte count            2  
Cycles                1  
Function              (ACC) - (CY) - #i8  
Flags affected        CY, AC, OV  
Interrupts enabled    Yes

### Description

Subtracts immediate data (i7 to i0) and carry flag from the contents of the accumulator, and stores the result in the accumulator.

### Example

		ACC	CY	AC	OV
MOV	#055H,ACC	55H	-	-	-
SUB	#013H	42H	0	0	0
SUBC	#003H	3FH	0	1	0
SUBC	#03FH	00H	0	0	0
SUBC	#002H	FEH	1	1	0
SUBC	#03EH	BFH	0	1	0

**SUBC d9**

**Subtract direct byte and carry flag from accumulator**

Instruction code	1 0 1 1 0 0 1d8 d7d6d5d4d3d2d1d0 (B2H to B3H)
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) - (CY) - (d9)
Flags affected	CY, AC, OV
Interrupts enabled	Yes

**Description**

Subtracts the contents of the RAM address or SFR specified by d8 to d0 and carry flag from the contents of the accumulator, and stores the result in the accumulator.

**Example 1**

		ACC	RAM	CY	AC	OV
						23H
MOV	#055H,ACC	55H	-	-	-	-
MOV	#068H,023H	55H	68H	-	-	-
SUB	#00CH	49H	68H	0	1	0
SUBC	023H	E1H	68H	1	0	0
SUBC	023H	78H	68H	0	1	1

**Example 2**

		ACC	B	CY	AC	OV
MOV	#080H,ACC	80H	-	-	-	-
MOV	#095H,B	80H	95H	-	-	-
SUB	#002H	7EH	95H	0	1	1
SUBC	B	E9H	95H	1	0	1
SUBC	B	53H	95H	0	0	1

## SUBC @Rj

Subtract indirect byte and carry flag from accumulator

Instruction code	1 0 1 1 0 1j1j0 (B4H to B7H)
Byte count	1
Cycles	1
Function	$(ACC) \leftarrow (ACC) - (CY) - ((Rj))$ j = 0, 1, 2, 3
Flags affected	CY, AC, OV
Interrupts enabled	Yes

## Description

Subtracts the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0 and carry flag from the contents of the accumulator, and stores the result in the accumulator.

### Example 1

		ACC	RAM	RAM	CY	AC	OV
			00H	68H			
MOV	#055H,ACC	55H	-	-	-	-	-
MOV	#068H,00H	55H	68H	-	-	-	-
MOV	#040H,@R0	55H	68H	40H	-	-	-
SUB	#016H	3FH	68H	40H	0	1	0
SUBC	@R0	FFH	68H	40H	1	0	0
SUBC	@R0	BEH	68H	40H	0	0	0

### Example 2

		ACC	RAM	TRL	CY	AC	OV
			02H				
MOV	#0AAH,ACC	AAH	-	-	-	-	-
MOV	#004H,002H	AAH	04H	-	-	-	-
MOV	#0AAH,@R2	AAH	04H	AAH	-	-	-
SUB	#001H	A9H	04H	AAH	0	0	0
SUBC	@R2	FFH	04H	AAH	1	1	0
SUBC	@R2	54H	04H	AAH	0	0	0

**INC d9****Increment direct byte**

Instruction code	0 1 1 0 0 0 1d8 d7d6d5d4d3d2d1d0 (62H to 63H)
Byte count	2
Cycles	1
Function	(d9) ← (d9) + 1
Flags affected	
Interrupts enabled	Yes

**Description**

Increments the contents of the RAM address or SFR specified by d8 to d0.

**Example 1**

		ACC
MOV	#0FDH, ACC	FDH
INC	ACC	FEH
INC	ACC	FFH
INC	ACC	00H
INC	ACC	01H

**Example 2**

		RAM
		7FH
MOV	#0FDH, 07FH	FDH
INC	07FH	FEH
INC	07FH	FFH
INC	07FH	00H
INC	07FH	01H

---

**Caution:**

- The flags, CY, AC, and OV are not changed.
- When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

---

## INC @Rj

### Increment indirect byte

Instruction code	0 1 1 0 0 1j1j0 (64H to 67H)
Byte count	1
Cycles	1
Function	$((Rj)) \leftarrow ((Rj)) + 1$ j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

### Description

Increments the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0.

### Example 1

		ACC	RAM
03H			
MOV	#000H,003H	-	00H
MOV	#0FDH,@R3	FDH	00H
INC	@R3	FEH	00H
INC	@R3	FFH	00H
INC	@R3	00H	00H

### Example 2

		RAM	RAM
		7FH	01H
MOV	#07FH,001H	-	7FH
MOV	#0FDH,@R1	FDH	7FH
INC	@R1	FEH	7FH
INC	@R1	FFH	7FH
INC	@R1	00H	7FH

---

**Caution:**

- The flags, CY, AC, and OV are not changed.
- When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

---

**DEC d9****Decrement direct byte**

Instruction code	0 1 1 1 0 0 1d8 d7d6d5d4d3d2d1d0 (72H to 73H)
Byte count	2
Cycles	1
Function	(d9) ← (d9) - 1
Flags affected	
Interrupts enabled	Yes

**Description**

Decrements the contents of the RAM address or SFR specified by d8 to d0.

**Example 1**

		ACC
MOV	#002H, ACC	02H
DEC	ACC	01H
DEC	ACC	00H
DEC	ACC	FFH
DEC	ACC	FEH

**Example 2**

		RAM
		7FH
MOV	#002H, 07FH	02H
DEC	07FH	01H
DEC	07FH	00H
DEC	07FH	FFH
DEC	07FH	FEH

---

**Caution:**

- The flags, CY, AC, and OV are not changed.
- When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

---

## DEC @Rj

### Decrement indirect byte

Instruction code	0 1 1 1 0 1j1j0 (74H to 77H)
Byte count	1
Cycles	1
Function	$((Rj)) \leftarrow ((Rj)) - 1$ j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

### Description

Decrements the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0.

### Example 1

		ACC	RAM
		02H	
MOV	#000H, 002H	-	00H
MOV	#002H, @R2	02H	00H
DEC	@R2	01H	00H
DEC	@R2	00H	00H
DEC	@R2	FFH	00H

### Example 2

		RAM	RAM
		7FH	00H
MOV	#07FH, 000H	-	7FH
MOV	#002H, @R0	02H	7FH
DEC	@R0	01H	7FH
DEC	@R0	00H	7FH
DEC	@R0	FFH	7FH

---

**Caution:**

- The flags, CY, AC, and OV are not changed.
- When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

---



**MUL**

**Multiply accumulator and C register by B register**

Instruction code    0 0 1 1 0 0 0 0 (30H)  
 Byte count            1  
 Cycles                7  
 Function               $(B)(ACC)(C) \leftarrow (ACC)(C) \times (B)$   
 Flags affected        CY, OV  
 Interrupts enabled    Yes on the 7th cycle

**Description**

Multiplies the unsigned 16-bit value represented by the accumulator and C register by the unsigned 8-bit value of the B register. Of the 24-bit calculation result, the bottom 8 bits are stored in C, the middle 8 bits in the accumulator, and the top 8 bits in B.

As a result of the calculation, if the contents of B are zero, the overflow flag is cleared, and if the contents of B are nonzero the overflow flag is set. The carry flag is always cleared.

**Example 1**

		ACC	C	B	CY	AC	OV
MOV	#0C4H,PSW	-	-	-	1	1	1
MOV	#011H,ACC	11H	-	-	1	1	1
MOV	#023H,C	11H	23H	-	1	1	1
MOV	#052H,B	11H	23H	52H	1	1	1
MUL		7DH	36H	05H	0	1	1

**Example 2**

		ACC	C	B	CY	AC	OV
MOV	#0C4H,PSW	-	-	-	1	1	1
MOV	#007H,ACC	07H	-	-	1	1	1
MOV	#005H,C	07H	05H	-	1	1	1
MOV	#010H,B	07H	05H	10H	1	1	1
MUL		70H	50H	00H	0	1	0

## DIV

### Divide accumulator and C register by B register

Instruction code	0 1 0 0 0 0 0 0 (40H)
Byte count	1
Cycles	7
Function	$(ACC)(C), \text{mod}(B) \leftarrow (ACC)(C) \div (B)$
Flags affected	CY, OV
Interrupts enabled	Yes on the 7th cycle

### Description

Divides the 16-bit value represented by the contents of the accumulator (upper byte) and C register (lower byte) by the contents of the B register (unsigned 8-bit value). The quotient is stored in the accumulator (upper byte) and C (lower byte), and the remainder is stored in B.

---

**Caution:** If this instruction is executed with the contents of the B register zero, the accumulator is set to FFH, and the overflow flag is set. If the B register is nonzero, then the overflow flag is cleared, and the carry flag is also always cleared.

---

### Example 1

		ACC	C	B	CY	AC	OV
MOV	#0C4H,PSW	-	-	-	1	1	1
MOV	#078H,ACC	79H	-	-	1	1	1
MOV	#005H,C	79H	05H	-	1	1	1
MOV	#007H,B	79H	05H	07H	1	1	1
DIV		11H	49H	06H	0	1	0

### Example 2

		ACC	C	B	CY	AC	OV
MOV	#0C0H,PSW	-	-	-	1	1	0
MOV	#007H,ACC	07H	-	-	1	1	0
MOV	#010H,C	07H	10H	-	1	1	0
MOV	#000H,B	07H	10H	00H	1	1	0
DIV		FFH	10H	00H	0	1	1 error

## Logical Instructions

### AND\_i8

#### AND immediate data to accumulator

Instruction code	1 1 1 0 0 0 1 i7i6i5i4i3i2i1i0 (E1H)
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) $\perp$ #i8
Flags affected	
Interrupts enabled	Yes

#### Description

Logical ANDs the contents of the accumulator and immediate data (i7 to i0), and stores the result in the accumulator.

#### Example 1

		ACC
MOV	#0FFH,ACC	FFH
AND	#0FAH	FAH
AND	#0AFH	AAH
AND	#00FH	0AH
AND	#0F0H	00H

#### Example 2

		ACC
MOV	#0FFH,ACC	FFH
AND	#0FEH	FEH
AND	#0FDH	FCH
AND	#0FBH	F8H
AND	#0F7H	F0H
AND	#0EFH	E0H
AND	#0DFH	C9H
AND	#0BFH	80H
AND	#07FH	00H

## AND d9

### AND direct byte to accumulator

Instruction code	1 1 1 0 0 0 1d8 d7d6d5d4d3d2d1d0 (E2H to E3H)
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) ∩ (d9)
Flags affected	
Interrupts enabled	Yes

### Description

Logical ANDs the contents of the accumulator and the contents of the RAM address or SFR specified by d8 to d0, and stores the result in the accumulator.

### Example 1

		ACC	RAM
			23H
MOV	#0FFH, ACC	FFH	-
MOV	#055H, 023H	FFH	55H
AND	023H	55H	55H
MOV	#0AAH, 023H	55H	AAH
AND	023H	00H	AAH

### Example 2

		ACC	B
MOV	#0FFH, ACC	FFH	-
MOV	#0FEH, B	FFH	FEH
AND	B	FEH	FEH
MOV	#0FDH, B	FEH	FDH
AND	B	FCH	FDH
MOV	#0FBH, B	FCH	FBH
AND	B	F8H	FBH
MOV	#0F7H, B	F8H	F7H
AND	B	F0H	F7H

**AND @Rj**

**AND indirect byte to accumulator**

Instruction code	1 1 1 0 0 1j1j0 (E4H to E7H)
Byte count	1
Cycles	1
Function	(ACC) ← (ACC) $\perp$ ((Rj)) j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

**Description**

Logical ANDs the contents of the accumulator and the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0, and stores the result in the accumulator.

**Example 1**

		ACC	RAM	RAM
			00H	68H
MOV	#0FFH,ACC	FFH	-	-
MOV	#068H,000H	FFH	68H	-
MOV	#0F0H,@R0	FFH	68H	F0H
AND	@R0	F0H	68H	F0H
MOV	#00FH,@R0	F0H	68H	0FH
AND	@R0	00H	68H	0FH

**Example 2**

		ACC	RAM	
			02H	TRL
MOV	#0FFH,ACC	FFH	-	-
MOV	#004H,002H	FFH	04H	-
MOV	#0EFH,@R2	FFH	04H	EFH
AND	@R2	EFH	04H	EFH
MOV	#0DFH,@R2	EFH	04H	DFH
AND	@R2	CFH	04H	DFH

## OR\_i8

### OR immediate data to accumulator

Instruction code	1 1 0 1 0 0 0 1 i7i6i5i4i3i2i1i0 (D1H)
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) ∩ #i8
Flags affected	
Interrupts enabled	Yes

### Description

Logical ORs the contents of the accumulator and immediate data (i7 to i0), and stores the result in the accumulator.

### Example 1

		ACC
MOV	#000H, ACC	00H
OR	#003H	03H
OR	#00CH	0FH
OR	#030H	3FH
OR	#0C0H	FFH

### Example 2

		ACC
MOV	#000H, ACC	00H
OR	#001H	01H
OR	#002H	03H
OR	#004H	07H
OR	#008H	0FH
OR	#010H	1FH
OR	#020H	3FH
OR	#040H	7FH
OR	#080H	FFH

**OR d9****OR direct byte to accumulator**

Instruction code	1 1 0 1 0 0 1d8 d7d6d5d4d3d2d1d0 (D2H to D3H)
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) ∨ (d9)
Flags affected	
Interrupts enabled	Yes

**Description**

Logical ORs the contents of the accumulator and the contents of the RAM address or SFR specified by d8 to d0, and stores the result in the accumulator.

**Example 1**

		ACC	RAM
			23H
MOV	#000H,ACC	00H	-
MOV	#055H,023H	00H	55H
OR	023H	55H	55H
MOV	#0AAH,023H	55H	AAH
OR	023H	FFH	AAH

**Example 2**

		ACC	B
MOV	#000H,ACC	00H	-
MOV	#001H,B	00H	01H
OR	B	01H	01H
MOV	#002H,B	01H	02H
OR	B	03H	02H
MOV	#004H,B	03H	04H
OR	B	07H	04H
MOV	#008H,B	07H	08H
OR	B	0FH	08H

## OR @Rj

### OR indirect byte to accumulator

Instruction code	1 1 0 1 0 1j1j0 (D4H to D7H)
Byte count	1
Cycles	1
Function	(ACC) ← (ACC) ⊞ ((Rj)) j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

### Description

Logical ORs the contents of the accumulator and the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0, and stores the result in the accumulator.

### Example 1

		ACC	RAM	RAM
			00H	68H
MOV	#000H, ACC	00H	-	-
MOV	#068H, 000H	00H	68H	-
MOV	#0F0H, @R0	00H	68H	F0H
OR	@R0	F0H	68H	F0H
MOV	#000FH, @R0	F0H	68H	0FH
OR	@R0	FFH	68H	0FH

### Example 2

		ACC	RAM	TRL
			02H	
MOV	#0AAH, ACC	AAH	-	-
MOV	#004H, 002H	AAH	04H	-
MOV	#005H, @R2	AAH	04H	05H
OR	@R2	AFH	04H	05H
MOV	#050H, @R2	AFH	04H	50H
OR	@R2	FFH	04H	50H



**XOR\_i8**

**XOR immediate data to accumulator**

Instruction code	1 1 0 1 0 1j1j0 (D4H to D7H)
Byte count	1
Cycles	1
Function	(ACC) ← (ACC) ⊖ ((Rj)) j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

**Description**

Logical XORs the contents of the accumulator and immediate data (i7 to i0), and stores the result in the accumulator.

**Example 1**

		ACC
MOV	#000H,ACC	00H
XOR	#00FH	0FH
XOR	#0F0H	FFH
XOR	#00FH	F0H
XOR	#0F0H	00H

**Example 2**

		ACC
MOV	#000H,ACC	00H
XOR	#001H	01H
XOR	#002H	03H
XOR	#004H	07H
XOR	#008H	0FH
XOR	#008H	07H
XOR	#004H	03H
XOR	#002H	01H
XOR	#001H	00H

## XOR d9

### XOR direct byte to accumulator

Instruction code	1 1 1 1 0 0 1d8 d7d6d5d4d3d2d1d0 (F2H to F3H)
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) ∨ (d9)
Flags affected	
Interrupts enabled	Yes

### Description

Logical XORs the contents of the accumulator and the contents of the RAM address or SFR specified by d8 to d0, and stores the result in the accumulator.

### Example 1

		ACC	RAM
		00H	23H
MOV	#000H,ACC	00H	-
MOV	#055H,023H	00H	55H
XOR	023H	55H	55H
MOV	#0FFH,023H	55H	FFH
XOR	023H	AAH	FFH

### Example 2

		ACC	B
		FFH	-
MOV	#0FFH,ACC	FFH	-
MOV	#010H,B	FFH	10H
XOR	B	EFH	10H
MOV	#020H,B	EFH	20H
XOR	B	CFH	20H
MOV	#040H,B	CFH	40H
XOR	B	8FH	40H
MOV	#080H,B	8FH	80H
XOR	B	0FH	80H

## XOR @Rj

### XOR indirect byte to accumulator

Instruction code	1 1 1 1 0 1j1j0 (F4H to F7H)
Byte count	1
Cycles	1
Function	$(ACC) \leftarrow (ACC) \vee ((Rj)) \quad j = 0, 1, 2, 3$
Flags affected	
Interrupts enabled	Yes

### Description

Logical XORs the contents of the accumulator and the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0, and stores the result in the accumulator.

### Example

		ACC	RAM	RAM
			01H	68H
MOV	#000H, ACC	00H	-	-
MOV	#068H, 001H	00H	68H	-
MOV	#0F0H, @R1	00H	68H	F0H
XOR	@R1	F0H	68H	F0H
MOV	#0FFH, @R1	F0H	68H	FFH
XOR	@R1	0FH	68H	FFH

### Example 2

		ACC	RAM	TRL
			03H	
MOV	#0AAH, ACC	AAH	-	-
MOV	#004H, 003H	AAH	04H	-
MOV	#0FFH, @R3	AAH	04H	FFH
XOR	@R3	55H	04H	FFH
XOR	@R3	AAH	04H	FFH
XOR	@R3	55H	04H	FFH
XOR	@R3	AAH	04H	FFH

## ROL

### Rotate accumulator left

Instruction code    1 1 1 0 0 0 0 0 (E0H)  
Byte count            1  
Cycles                1  
Function               $\leftarrow A7 \leftarrow A6 \leftarrow A5 \leftarrow A4 \leftarrow A3 \leftarrow A2 \leftarrow A1 \leftarrow A0 \leftarrow (A7)$   
Flags affected  
Interrupts enabled    Yes

### Description

Rotates the 8-bit value of the accumulator left by one bit position. This transfers bit 7 of the accumulator to bit 0.

### Example 1

		ACC		
MOV	#01H, ACC	01H	0000	0001B
ROL		02H	0000	0010B
ROL		04H	0000	0100B
ROL		08H	0000	1000B
ROL		10H	0001	0000B
ROL		20H	0010	0000B
ROL		40H	0100	0000B
ROL		80H	1000	0000B
ROL		01H	0000	0001B
MOV	#55H, ACC	55H	0101	0101B
ROL		AAH	1010	1010B
ROL		55H	0101	0101B
ROL		AAH	1010	1010B
ROL		55H	0101	0101B

**ROL**

**Rotate accumulator left through the carry flag**

Instruction code     1 1 1 1 0 0 0 0 (F0H)  
 Byte count             1  
 Cycles                 1  
 Function               ←A7←A6←A5←A4←A3←A2←A1←A0←CY← (A7)  
 Flags affected         CY  
 Interrupts enabled     Yes

**Description**

Rotates the 8-bit value of the accumulator left by one bit position through the carry flag. This transfers bit 7 of the accumulator to the carry flag, and the contents of the carry flag to bit 0.

**Example 1**

		ACC		CY	
MOV	#01H, ACC	01H	000 0	0001B	-
SET1	PSW, 7	01H	0000	001B	1
ROL		03H	0000	0011B	0
ROL		06H	0000	0110B	0
ROL		0CH	0000	1100B	0
ROL		11H	0001	1000B	0
ROL		30H	0011	0000B	0
ROL		60H	0110	0000B	0
ROL		C0H	1100	0000B	0
ROL		80H	1000	0000B	1
ROL		01H	0000	0001B	1
MOV	#55H, ACC	55H	0101	0101B	1
ROL		ABH	1010	1011B	0
ROL		56H	0101	0110B	1
ROL		ADH	1010	1101B	0

## ROR

### Rotate accumulator right

Instruction code	1 1 0 0 0 0 0 0 (C0H)
Byte count	1
Cycles	1
Function	(A0) →A7→A6→A5→A4→A3→A2→A1→A0→
Flags affected	
Interrupts enabled	Yes

### Description

Rotates the 8-bit value of the accumulator right by one bit position. This transfers bit 0 of the accumulator to bit 7.

### Example 1

		ACC		
MOV	#01H, ACC	01H	0000	0001B
ROR		80H	1000	0000B
ROR		40H	0100	0000B
ROR		20H	0010	0000B
ROR		10H	0001	0000B
ROR		08H	0000	1000B
ROR		04H	0000	0100B
ROR		02H	0000	0010B
ROR		01H	0000	0001B
MOV	#51H, ACC	51H	0101	0001B
ROR		A8H	1010	1000B
ROR		54H	0101	0100B
ROR		2AH	0010	1010B
ROR		15H	0001	0101B

**RORC**

**Rotate accumulator right through the carry flag**

Instruction code    1 1 0 1 0 0 0 0 (D0H)  
 Byte count            1  
 Cycles                1  
 Function              (A0) →CY→A7→A6→A5→A4→A3→A2→A1→A0→  
 Flags affected        CY  
 Interrupts enabled    Yes

**Description**

Rotates the 8-bit value of the accumulator right by one bit position through the carry flag. This transfers bit 0 of the accumulator to the carry flag, and the contents of the carry flag to bit 7.

**Example 1**

		ACC			CY
MOV	#01H, ACC	01H	0000	0001B	-
SET1	PSW, 7	01H	0000	0001B	1
RORC		80H	1000	0000B	1
RORC		C0H	1100	0000B	0
RORC		60H	0110	0000B	0
RORC		30H	0011	0000B	0
RORC		18H	0001	1000B	0
RORC		0CH	0000	1100B	0
RORC		06H	0000	0110B	0
RORC		03H	0000	0011B	0
RORC		01H	0000	0001B	1
MOV	#55H, ACC	55H	0101	0101B	1
RORC		AAH	1010	1010B	1
RORC		D5H	1101	0101B	0
RORC		6AH	0110	1010B	1

## Data Transfer Instructions

### LD d9

#### Load direct byte to accumulator

Instruction code	0 0 0 0 0 0 1d8 d8d7d6d5d4d3d2d1d0 (02H to 03H)
Byte count	2
Cycles	1
Function	(ACC) ← (d9)
Flags affected	
Interrupts enabled	Yes

#### Description

Transfers the contents of the RAM address or SFR specified by d8 to d0 to the accumulator.

#### Example 1

		ACC	RAM	RAM
			70H	71H
MOV	#0FF,ACC	FFH	-	-
MOV	#055H,070H	FFH	55H	-
MOV	#0AAH,071H	FFH	55H	AAH
LD	070H	55H	55H	AAH
LD	071H	AAH	55H	AAH

#### Example 2

		ACC	B	SP
MOV	#0FF,ACC	FFH	-	-
MOV	#0F0H,B	FFH	F0H	-
MOV	#00FH,SP	FFH	F0H	0FH
LD	B	F0H	F0H	0FH
LD	SP	0FH	F0H	0FH
LD	B	F0H	F0H	0FH



**LD @Rj**

**Load indirect byte to accumulator**

Instruction code	0 0 0 0 0 1j1j0 (04H to 07H)
Byte count	1
Cycles	1
Function	(ACC) ← ((Rj)) j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

**Description**

Transfers the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0 to the accumulator.

**Example 1**

		ACC	RAM	RAM	RAM	RAM
			00H	01H	70H	7FH
MOV	#0FFH,ACC	FFH	-	-	-	-
MOV	#070H,000H	FFH	70H	-	-	-
MOV	#07FH,001H	FFH	70H	7FH	-	-
MOV	#0F0H,@R0	FFH	70H	7FH	F0H	-
MOV	#00FH,@R1	FFH	70H	7FH	F0H	0FH
LD	@R0	F0H	70H	7FH	F0H	0FH
LD	@R1	0FH	70H	7FH	F0H	0FH

**Example 2**

		ACC	RAM	RAM	B	C
			02H	03H	102H	103H
MOV	#0FF,ACC	FFH	-	-	-	-
MOV	#004H,002H	FFH	04H	-	-	-
MOV	#005H,003H	FFH	04H	05H	-	-
MOV	#0AAH,@R2	FFH	04H	05H	AAH	-
MOV	#055H,@R3	FFH	04H	05H	AAH	55H
LD	@R2	AAH	04H	05H	AAH	55H
LD	@R3	55H	04H	05H	AAH	55H

## ST d9

### Store direct byte from accumulator

Instruction code	0 0 0 1 0 0 1d8 d7d6d5d4d3d2d1d0 (12H to 13H)
Byte count	2
Cycles	1
Function	(d9) ← (ACC)
Flags affected	
Interrupts enabled	Yes

### Description

Transfers the contents of the accumulator to the RAM address or special function register (SFR) specified by d8 to d0.

### Example 1

		ACC	RAM	RAM
			70H	71H
MOV	#0FFH,ACC	FFH	-	-
MOV	#055H,070H	FFH	55H	-
MOV	#0AAH,071H	FFH	55H	AAH
ST	070H	FFH	FFH	AAH
MOV	#000H,ACC	00H	FFH	AAH
ST	071H	00H	FFH	00H

### Example 2

		ACC	B	SP
MOV	#012H,ACC	12H	-	-
MOV	#0F0H,B	12H	F0H	-
MOV	#00FH,SP	12H	F0H	0FH
ST	B 12H	12H	0FH	
MOV	#034H,ACC	34H	12H	0FH
ST	SP	34H	12H	34H
ST	B	34H	34H	34H

**ST @Rj**

**Store indirect byte from accumulator**

Instruction code	0 0 0 1 0 1j1j0 (14H to 17H)
Byte count	1
Cycles	1
Function	((Rj)) ← (ACC) j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

**Description**

Transfers the contents of the accumulator to the RAM address or special function register (SFR) specified by the indirect address register specified by j1 to j0.

**Example 1**

		ACC	RAM	RAM	RAM	RAM
			00H	01H	70H	7FH
MOV	#0FFH,ACC	FFH	-	-	-	-
MOV	#070H,000H	FFH	70H	-	-	-
MOV	#07FH,001H	FFH	70H	7FH	-	-
MOV	#0F0H,@R0	FFH	70H	7FH	F0H	-
MOV	#00FH,@R1	FFH	70H	7FH	F0H	0FH
ST	@R0	FFH	70H	7FH	FFH	0FH
ST	@R1	FFH	70H	7FH	FFH	FFH

**Example 2**

		ACC	RAM	RAM	TRL	TRH
			02H	03H	104H	105H
MOV	#000H,ACC	00H	-	-	-	-
MOV	#004H,002H	00H	04H	-	-	-
MOV	#005H,003H	00H	04H	05H	-	-
MOV	#0AAH,@R2	00H	04H	05H	AAH	-
MOV	#055H,@R3	00H	04H	05H	AAH	55H
ST	@R2	00H	04H	05H	00H	55H
ST	@R3	00H	04H	05H	00H	00H

## MOV\_i8, d9

### Move immediate data to direct byte

Instruction code    0 0 1 0 0 0 1d8 d7d6d5d4d3d2d1d0 i7i6i5i4i3i2i1i0 (22H to 23H)  
Byte count            3  
Cycles                2  
Function              (d9) ← #i8  
Flags affected  
Interrupts enabled    Yes on the 2nd cycle

### Description

Transfers immediate data (i7 to i0) to the RAM address or special function register (SFR) specified by d8 to d0.

### Example 1

		RAM	RAM	RAM	RAM
		00H	01H	02H	03H
MOV	#0FFH,000H	FFH	-	-	-
MOV	#0FEH,001H	FFH	FEH	-	-
MOV	#0FDH,002H	FFH	FEH	FDH	-
MOV	#0FCH,003H	FFH	FEH	FDH	FCH
MOV	#0FBH,003H	FFH	FEH	FDH	FBH
MOV	#0FAH,002H	FFH	FEH	FAH	FBH
MOV	#0F9H,001H	FFH	F9H	FAH	FBH
MOV	#0F8H,000H	F8H	F9H	FAH	FBH

### Example 2

		ACC	B	TRL
MOV	#0FFH,100H	FFH	-	-
MOV	#0FEH,102H	FFH	FEH	-
MOV	#0FDH,104H	FFH	FEH	FDH
MOV	#0FAH,104H	FFH	FEH	FAH
MOV	#0F9H,102H	FFH	F9H	FAH
MOV	#0F8H,100H	F8H	F9H	FAH

**MOV \_i8, @Rj**

**Move immediate data to indirect byte**

Instruction code     0 0 1 0 0 1j1j0 i7i6i5i4i3i2i1i0 (24H to 27H)  
 Byte count             2  
 Cycles                 1  
 Function               ((Rj)) ← #i8 j = 0, 1, 2, 3  
 Flags affected  
 Interrupts enabled     Yes

**Description**

Transfers immediate data (i7 to i0) to the RAM address or special function register (SFR) specified by the indirect address register specified by j1 to j0.

**Example 1**

		RAM	RAM	RAM	RAM
		00H	01H	7EH	7FH
MOV	#07FH,000H	7FH	-	-	-
MOV	#07EH,001H	7FH	7EH	-	-
MOV	#0FDH,@R0	7FH	7EH	-	FDH
MOV	#0FCH,@R1	7FH	7EH	FCH	FDH
MOV	#0FBH,@R0	7FH	7EH	FCH	FBH
MOV	#0FAH,@R1	7FH	7EH	FAH	FBH
MOV	#0F9H,@R0	7FH	7EH	FAH	F9H
MOV	#0F8H,@R1	7FH	7EH	F8H	F9H

**Example 2**

		RAM	RAM	ACC	B
		02H	03H	100H	102H
MOV	#000H,002H	00H	-	-	-
MOV	#002H,003H	00H	02H	-	-
MOV	#0FDH,@R2	00H	02H	FDH	-
MOV	#0FCH,@R3	00H	02H	FDH	FCH
MOV	#0FBH,@R2	00H	02H	FBH	FCH
MOV	#0FAH,@R3	00H	02H	FBH	FAH

## LDC

### Load code byte relative to TRR to accumulator

Instruction code    1 1 0 0 0 0 1 (C1H)  
Byte count            1  
Cycles                2  
Function              (ACC) ← (BNK)((TRR) + (ACC)) [ROM]  
Flags affected  
Interrupts enabled    Yes on the 2nd cycle

### Description

Loads into the accumulator the contents of the program memory (ROM) address specified by adding the contents of the accumulator to the contents of the table reference register (TRR). The ROM address accessed is different for a program running in ROM and a program running in flash memory. For a program running in ROM, ROM is accessed, and for a program running in flash memory, bank 0 of flash memory is accessed.

The LDC instruction cannot access bank 1 of flash memory. To access bank 1 of flash memory, use the system BIOS function provided by Visual Memory.

---

**Reference:** For details of operating system function calls, refer to the "System BIOS" section of the Visual Memory Hardware Manual.

---

### Example

		ACC	TRR	TRR	TRR
			TRH	TRL	+ACC
MOV	#001H,TRH	-	01H	-	-
MOV	#023H,TRL	-	01H	23H	-
MOV	#000H,ACC	00H	01H	23H	0123H
LDC		30H	01H	23H	0153H
MOV	#001H,ACC	01H	01H	23H	0124H
LDC		FFH	01H	23H	0222H
MOV	#002H,ACC	02H	01H	23H	0125H
LDC		57H	01H	23H	017AH
MOV	#003H,ACC	03H	01H	23H	0126H
LDC		EAH	01H	23H	020DH
		PC	ROM		
		0123H	30H		
		0124H	FFH		
		0125H	57H		
		0126H	EAH		

**PUSH d9**

**Push direct byte to stack**

Instruction code    0 1 1 0 0 0 0d8 d7d6d5d4d3d2d1d0 (60H to 61H)  
 Byte count            2  
 Cycles                2  
 Function               $(SP) \leftarrow (SP) + 1, ((SP)) \leftarrow (d9)$   
 Flags affected  
 Interrupts enabled    Yes on the 2nd cycle

**Description**

Increments the stack pointer (SP), then transfers the contents of the RAM address or SFR specified by d8 to d0 to the address indicated by the stack pointer.

**Note:** Even when the application is accessing RAM bank 1, the stack is in RAM bank 0.

**Example**

		ACC	B	RAM	SP	RAM	RAM	RAM
				00H		20H	21H	22H
MOV	#0AAH,ACC	AAH	-	-	-	-	-	-
MOV	#055H,B	AAH	55H	-	-	-	-	-
MOV	#012H,000H	AAH	55H	12H	-	-	-	-
MOV	#01FH,SP	AAH	55H	12H	1FH	-	-	-
PUSH	ACC	AAH	55H	12H	20H	AAH	-	-
PUSH	B	AAH	55H	12H	21H	AAH	55H	-
PUSH	000H	AAH	55H	12H	22H	AAH	55H	12H
POP	B	AAH	12H	12H	21H	AAH	55H	12H
POP	ACC	55H	12H	12H	20H	AAH	55H	12H
POP	000H	55H	12H	AAH	1FH	AAH	55H	12H

## POP d9

### Pop direct byte from stack

Instruction code	0 1 1 1 0 0 0d8 d7d6d5d4d3d2d1d0 (70H to 71H)
Byte count	2
Cycles	2
Function	(d9) ← ((SP)), (SP) ← (SP) - 1
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

### Description

Transfers the contents of the RAM address indicated by the stack pointer to the RAM address or special function register (SFR) specified by d8 to d0, then decrements the stack pointer.

---

**Note:** Even when the application is accessing RAM bank 1, the stack is in RAM bank 0.

---

### Example

		ACC	B	TRL	SP	RAM 20H	RAM 21H	RAM 22H
MOV	#0AAH,ACC	AAH	-	-	-	-	-	-
MOV	#055H,B	AAH	55H	-	-	-	-	-
MOV	#012H,TRL	AAH	55H	12H	-	-	-	-
MOV	#01FH,SP	AAH	55H	12H	1FH	-	-	-
PUSH	ACC	AAH	55H	12H	20H	AAH	-	-
PUSH	B	AAH	55H	12H	21H	AAH	55H	-
PUSH	TRL	AAH	55H	12H	22H	AAH	55H	12H
POP	B	AAH	12H	12H	21H	AAH	55H	12H
POP	ACC	55H	12H	12H	20H	AAH	55H	12H
POP	TRL	55H	12H	AAH	1FH	AAH	55H	12H



**XCH d9**

**Exchange direct byte with accumulator**

Instruction code	1 1 0 0 0 0 1d8 d7d6d5d4d3d2d1d0 (C2H to C3H)
Byte count	2
Cycles	1
Function	(ACC) $\longleftrightarrow$ (d9)
Flags affected	
Interrupts enabled	Yes

**Description**

Exchanges the contents of the accumulator and the contents of the RAM address or SFR specified by d8 to d0.

**Example 1**

		ACC	RAM
			23H
MOV	#0FFH,ACC	FFH	-
MOV	#055H,023H	FFH	55H
XCH	023H	55H	FFH
XCH	023H	FFH	55H
XCH	023H	55H	FFH
XCH	023H	FFH	55H

**Example 2**

		ACC	B
MOV	#0FFH,ACC	FFH	-
MOV	#0FEH,B	FFH	FEH
XCH	B	FEH	FFH
XCH	B	FFH	FEH
XCH	B	FEH	FFH
XCH	B	FFH	FEH

## XCH @Rj

### Exchange indirect byte with accumulator

Instruction code	1 1 0 0 0 1j1j0 (C4H to C7H)
Byte count	1
Cycles	1
Function	(ACC) $\leftrightarrow$ ((Rj)) j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

### Description

Exchanges the contents of the accumulator and the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0.

### Example 1

		ACC	RAM	RAM
			01H	68H
MOV	#0FFH, ACC	FFH	-	-
MOV	#068H, 001H	FFH	68H	-
MOV	#0F0H, @R1	FFH	68H	F0H
XCH	@R1	F0H	68H	FFH
XCH	@R1	FFH	68H	F0H
XCH	@R1	F0H	68H	FFH
XCH	@R1	FFH	68H	F0H

### Example 2

		ACC	RAM	TRL
			03H	
MOV	#0AAH, ACC	AAH	-	-
MOV	#004H, 003H	AAH	04H	-
MOV	#055H, @R3	AAH	04H	55H
XCH	@R3	55H	04H	AAH
XCH	@R3	AAH	04H	55H
XCH	@R3	55H	04H	AAH

# Jump Instructions

## **JMP a12**

### **Jump near absolute address**

Instruction code	0 0 1a11 1a10a9a8 a7a6a5a4a3a2a1a0 (28H to 2FH, 38H to 3FH)
Byte count	2
Cycles	2
Function	(PC) ← (PC) + 2, (PC11 to 00) ← a12
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

### **Description**

Increments the program counter (PC) twice, then transfers the value of a11 to a0 to PC bits 11 to 00.

### **Example 1**

The value of label LA is 0F0EH.

		PC	Instruction code
	NOP	0FFBH	00H
	NOP	0FFCH	00H
	JMP	0FFDH	3F0EH
LA:	INC	0F0EH	6300H
	ROR	0F10H	C0H

### **Example 2**

The value of label LA is 1F0EH.

		PC	Instruction code
	NOP	0FFCH	00H
	NOP	0FFDH	00H
	JMP	0FFEH	3F0EH
LA:	INC	1F0EH	6300H
	ROR	1F10H	C0H

## JMPF a16

### Jump far absolute address

Instruction code    0 0 1 0 0 0 0 1 a15a14a13a12a11a10a9a8 a7a6a5a4a3a2a1a0 (21H)  
Byte count                    3  
Cycles                         2  
Function                        (PC) ← a16  
Flags affected  
Interrupts enabled              Yes on the 2nd cycle

### Description

Transfers the value of a15 to a0 to the program counter (PC).

### Example 1

The value of label LA is 0F0EH.

		PC	Instruction code
	NOP	0FFAH	00H
	NOP	0FFBH	00H
	JMPF                    LA	0FFCH	210F0EH
LA:	INC                     ACC	0F0EH	6300H
	ROR	0F10H	C0H

### Example 2

The value of label LA is 0F0EH.

		PC	Instruction code
	NOP	0FFCH	00H
	NOP	0FFDH	00H
	JMPF                    LA	0FFEH	210F0EH
LA:	INC                     ACC	0F0EH	6300H
	ROR	0F10H	C0H

**BR r8**

**Branch near relative address**

Instruction code    0 0 0 0 0 0 0 1 r7r6r5r4r3r2r1r0 (01H)  
 Byte count                    2  
 Cycles                         2  
 Function                         $(PC) \leftarrow (PC) + 2, (PC) \leftarrow (PC) + r8$   
 Flags affected  
 Interrupts enabled            Yes on the 2nd cycle

**Description**

Increments the program counter (PC) twice, then adds the value of r7 to r0 to the PC, leaving the result in PC.

**Example 1**

The value of label LA is 0F5FH.

			PC	Instruction code
	NOP		0F1CH	00H
	NOP		0F1DH	00H
	BR	LA	0F1EH	013FH
LA:	INC	ACC	0F5FH	6300H
	ROR		0F61H	C0H

**Example 2**

The value of label LA is 1F0EH.

			PC	Instruction code
	NOP		1F0CH	00H
	NOP		1F0DH	00H
LA:	INC	ACC	1F0EH	6300H
	ROR		1F10H	C0H
	NOP		1F11H	00H
	NOP		1F12H	00H
	BR	LA	1F13H	01F9H

## BRF r16

### Branch far relative address

Instruction code    0 0 0 1 0 0 0 1 r7r6r5r4r3r2r1r0 r15r14r13r12r11r10r9r8 (11H)  
Byte count                    3  
Cycles                         4  
Function                         $(PC) \leftarrow (PC) + 3, (PC) \leftarrow (PC) - 1 + r16$   
Flags affected  
Interrupts enabled            Yes on the 4th cycle

### Description

Increments the program counter (PC) three times, then decrements PC and further adds the value of r15 to r0 to PC, leaving the result in PC.

### Example 1

The value of label LA is 105FH.

		PC	Instruction code
	NOP	0F1CH	00H
	NOP	0F1DH	00H
	BRF	0F1EH	113F01H
LA:	INC	105FH	6300H
	ROR	1061H	C0H

### Example 2

The value of label LA is 1F0EH.

		PC	Instruction code
	NOP	1FFCH	00H
	NOP	1FFDH	00H
LA:	INC	1F0EH	6300H
	ROR	1F10H	C0H
	NOP	1F11H	00H
	NOP	1F12H	00H
	BRF	1F13H	11F8FFH

## Conditional Branch Instructions

### **BZ r8**

#### **Branch near relative address if accumulator is zero**

Instruction code	1 0 0 0 0 0 0 r7r6r5r4r3r2r1r0 (80H)
Byte count	2
Cycles	2
Function	(PC) ← (PC) + 2, if (ACC) = 0 then (PC) ← (PC) + r8
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

#### **Description**

Increments the program counter (PC) twice, then if the accumulator is zero, adds the value of r7 to r0 to PC, leaving the result in PC.

If the accumulator is nonzero, continues to the next instruction.

#### **Example 1**

When the BZ instruction is executed, the accumulator is zero, so control branches to label LA.

		PC	Instruction code	ACC
	MOV	#000H,ACC	0F1BH 230000H	00H
	BZ	LA	0F1EH 803FH	00H
LA:	INC	ACC	0F5FH 6300H	01H
	ROR		0F61H C0H	80H

#### **Example 2**

When the BZ instruction is executed, the accumulator is nonzero, so control passes to the next instruction.

		PC	Instruction code	ACC
	MOV	#001H,ACC	0F1BH 230001H	01H
	BZ	LA	0F1EH 803FH	01H
	DEC	ACC	0F20H 7300H	00H
	ROR		0F22H C0H	00H
LA:	INC	ACC		

## BNZ r8

### Branch near relative address if accumulator is not zero

Instruction code    1 0 0 1 0 0 0 0 r7r6r5r4r3r2r1r0 (90H)  
Byte count                    2  
Cycles                         2  
Function                         $(PC) \leftarrow (PC) + 2$ , if  $(ACC) \neq 0$  then  $(PC) \leftarrow (PC) + r8$   
Flags affected  
Interrupts enabled            Yes on the 2nd cycle

### Description

Increments the program counter (PC) twice, then if the accumulator is nonzero, adds the value of r7 to r0 to PC, leaving the result in PC.

If the accumulator is zero, continues to the next instruction.

### Example 1

When the BNZ instruction is executed, the accumulator is nonzero, so control branches to label LA.

			PC	Instruction code	ACC
	MOV	#001H,ACC	0F1BH	230001H	01H
BNZ	LA	0F1EH	903FH	01H	
LA:	INC	ACC	0F5FH	6300H	02H
	ROR		0F61H	C0H	01H

### Example 2

When the BNZ instruction is executed, the accumulator is zero, so control passes to the next instruction.

			PC	Instruction code	ACC
	MOV	#000H,ACC	0F1BH	230000H	00H
	BNZ	LA	0F1EH	903FH	00H
	DEC	ACC	0F20H	7300H	FFH
	ROR		0F22H	C0H	FFH
LA:	INC	ACC			



**BP d9, b3, r8**

**Branch near relative address if direct bit is one ("positive")**

Instruction code	0 1 1d8 1b2b1b0 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 (68H to 6FH, 78H to 7FH)
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, if (d9, b3) = 1 then (PC) ← (PC) + r8
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

**Description**

Increments the program counter (PC) three times, then if the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0 is set (1), adds the value of r7 to r0 to PC, leaving the result in PC.

If the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0 is cleared (0), continues to the next instruction.

**Example 1**

When the BP instruction is executed, bit 0 of B is 1, so control branches to the label LA.

		PC	Instruction code	B
	MOV	#001H,B	0F1AH 230201H	01H
	BP	B,0,LA	0F1DH 78023FH	01H
LA:	INC	B	0F5FH 6302H	02H
	NOP		0F61H 00H	02H

**Example 2**

When the BP instruction is executed, bit 0 of the accumulator is 0, so control passes to the next instruction.

		PC	Instruction code	ACC
	MOV	#080H,ACC	0F1AH 230080H	80H
	BP	ACC,0,LA	0F1DH 78003FH	80H
	DEC	ACC	0F20H 7300H	7FH
	ROR		0F22H C0H	BFH
LA:	INC	ACC		

## BPC d9, b3, r8

Branch near relative address if direct bit is one ("positive"), and clear

Instruction code	0 1 0d8 1b2b1b0 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 (48H to 4FH, 58H to 5FH)
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, if (d9, b3) = 1 then (PC) ← (PC) + r8, (d9, b3) = 0
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

## Description

Increments the program counter (PC) three times, then if the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0 is set (1), first clears the bit, then adds the value of r7 to r0 to PC, leaving the result in PC.

If the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0 is cleared (0), continues to the next instruction.

## Example 1

When the BPC instruction is executed, bit 0 of B is 1, so it is cleared, then control branches to the label LA.

		PC	Instruction code	B
	MOV	#003H,B	0F1AH 230203H	03H
	BPC	B,0,LA	0F1DH 58023FH	02H
LA:	INC	B	0F5FH 6302H	03H
	NOF		0F61H 00H	03H

## Example 2

When the BPC instruction is executed, bit 0 of the accumulator is 0, so control passes to the next instruction.

		PC	Instruction code	ACC
	MOV	#080H,ACC	0F1AH 230080H	80H
	BPC	ACC,0,LA	0F1DH 58003FH	80H
	DEC	ACC	0F20H 7300H	7FH
	ROR		0F22H C0H	BFH
LA:	INC	ACC		

---

**Caution:** When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

---

**BN d9, b3, r8**

**Branch near relative address if direct bit is zero ("negative")**

Instruction code	1 0 0d8 1b2b1b0 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 (88H to 8FH, 98H to 9FH)
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, if (d9, b3) = 0 then (PC) ← (PC) + r8
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

**Description**

Increments the program counter (PC) three times, then if the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0 is cleared (0), adds the value of r7 to r0 to PC, leaving the result in PC.

If the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0 is set (1), continues to the next instruction.

**Example 1**

When the BN instruction is executed, bit 0 of B is zero, so control branches to the label LA.

		PC	Instruction code	B
	MOV	#0FEH, B	0F1AH 2302FEH	FEH
	BN	B, 0, LA	0F1DH 98023FH	FEH
LA:	INC	B	0F5FH 6302H	FFH
	NOP		0F61H 00H	FFH

**Example 2**

When the BN instruction is executed, bit 0 of the accumulator is 1, so control passes to the next instruction.

		PC	Instruction code	ACC
	MOV	#001H, ACC	0F1AH 230001H	01H
	BN	ACC, 0, LA	0F1DH 98003FH	01H
	DEC	ACC	0F20H 7300H	00H
	ROR		0F22H C0H	00H
LA:	INC	ACC		

## DBNZ d9, r8

### Decrement direct byte and branch near relative address if direct byte is nonzero

Instruction code	0 1 0 1 0 0 1d8 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 (52H to 53H)
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, (d9) = (d9)-1, if (d9) ≠ 0 then (PC) ← (PC) + r8
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

### Description

Increments the program counter (PC) three times, then decrements the address in RAM or special function register (SFR) indicated by d8 to d0. Next, if the value of the RAM address or SFR after decrementing is nonzero, adds the value of r7 to r0 to PC, leaving the result in PC.

If the value of the RAM address or SFR after decrementing is zero, continues to the next instruction.

### Example 1

When the DBNZ instruction is executed, B is decremented, and since B is then nonzero, control branches to the label LA.

		PC	Instruction code	B
	MOV	#002H,B	0F1AH 230202H	02H
	DBNZ	B,LA	0F1DH 53023FH	01H
LA:	INC	B	0F5FH 6302H	02H
	NOP		0F61H 00H	02H

### Example 2

When the DBNZ instruction is executed, the accumulator is decremented, and since the accumulator is then zero, control passes to the next instruction.

		PC	Instruction code	ACC
	MOV	#001H,ACC	0F1AH 230001H	01H
	DBNZ	ACC,LA	0F1DH 53003FH	00H
	DEC	ACC	0F20H 7300H	FFH
	ROR		0F22H C0H	FFH
LA:	INC	ACC		

---

**Caution:** When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

---

**DBNZ @Rj, r8**

**Decrement indirect byte and branch near relative address if indirect byte is not zero**

Instruction code	0 1 0 1 0 1j1j0 r7r6r5r4r3r2r1r0 (54H to 57H)
Byte count	2
Cycles	2
Function	(PC) ← (PC) + 2, ((Rj)) = ((Rj)) - 1, if ((Rj)) ≠ 0 then (PC) ← (PC) + r8 j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

**Description**

Increments the program counter (PC) twice, then decrements the address in RAM or special function register (SFR) indicated by the indirect address register specified by j1 to j0. Next, if the value of the RAM address or SFR after decrementing is nonzero, adds the value of r7 to r0 to PC, leaving the result in PC. If the value of the RAM address or SFR after decrementing is zero, continues to the next instruction.

**Example 1**

When the DBNZ instruction is executed, B is decremented, and since B is then nonzero, control branches to the label LA.

			PC	Instruction code	B	RAM
						03H
	MOV	#002H, B	0F18H	230202H	02H	-
	MOV	#002H, 003H	0F1BH	220302H	02H	02H
	DBNZ	@R3, LA	0F1EH	573FH	01H	02H
LA:	INC	B	0F5FH	6302H	02H	02H

**Example 2**

When the DBNZ instruction is executed, the accumulator is decremented, and since the accumulator is then zero, control passes to the next instruction.

			PC	Instruction code	ACC	RAM
						03H
	MOV	#001H, ACC	0F18H	230001H	01H	-
	MOV	#000H, 003H	0F1BH	220300H	01H	00H
	DBNZ	@R3, LA	0F1EH	573FH	00H	00H
	DEC	ACC	0F20H	7300H	FFH	00H
LA:	INC	ACC				

**Caution:** When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

## BE\_i8, r8

### Compare immediate data to accumulator and branch near relative address if equal

Instruction code	0 0 1 1 0 0 0 1	i7i6i5i4i3i2i1i0	r7r6r5r4r3r2r1r0	(31H)
Byte count	3			
Cycles	2			
Function	(PC) ← (PC) + 3, if (ACC) = #i8 then (PC) ← (PC) + r8			
Flags affected	CY			
Interrupts enabled	Yes on the 2nd cycle			

### Description

Increments the program counter (PC) three times, then compares immediate data (i7 to i0) with the contents of the accumulator, and if the values are equal adds the value of r7 to r0 to PC, leaving the result in PC.

If the values are different, continues to the next instruction.

If the value in the accumulator is less than the immediate data value, the carry flag is set; if equal or more, the carry flag is cleared.

$ACC < \#i8 \rightarrow CY = 1$

$ACC \geq \#i8 \rightarrow CY = 0$

### Example 1

When the BE instruction is executed, ACC=02H, so CY is cleared, and control branches to the label LA.

			PC	Instruction code	ACC	CY
	MOV	#002H,ACC	0F1AH	230002H	02H	-
	BE	#002H,LA	0F1DH	31023FH	02H	0
LA:	INC	ACC		0F5FH 6300H	03H	0

### Example 2

When the BE instruction is executed, ACC< 04H, so CY is set, and control passes to the next instruction.

			PC	Instruction code	ACC	CY
	MOV	#003H,ACC	0F1AH	230003H	03H	-
	BE	#004H,LA	0F1DH	31043FH	03H	1
	DEC	ACC	0F20H	7300H	02H	1
LA:	INC	ACC				

**BE d9, r8**

**Compare direct byte to accumulator and branch near relative address if equal**

Instruction code	0 0 1 1 0 0 1d8 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 (32H to 33H)
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, if (ACC) = (d9) then (PC) ← (PC) + r8
Flags affected	CY
Interrupts enabled	Yes on the 2nd cycle

**Description**

Increments the program counter (PC) three times, then compares the contents of the RAM address or SFR specified by d8 to d0 with the contents of the accumulator, and if the values are equal adds the value of r7 to r0 to PC, leaving the result in PC.

If the values are different, continues to the next instruction.

If the value in the accumulator is less than the contents of the RAM address or SFR specified by d8 to d0, the carry flag is set; if equal or more, the carry flag is cleared.

ACC < d9 (RAM or SFR) → CY = 1

ACC ≥ d9 (RAM or SFR) → CY = 0

**Example 1**

When the BE instruction is executed, ACC=B, so CY is cleared, and control branches to the label LA.

			PC	Instruction code	ACC	B	CY
	MOV	#002H, ACC	0F17H	230002H	02H	-	-
	MOV	#002H, B	0F1AH	230202H	02H	02H	-
	BE	B, LA	0F1DH	33023FH	02H	02H	0
LA:	INC	ACC	0F5FH	6300H	03H	02H	0

**Example 2**

When the BE instruction is executed, ACC=02H, so CY is set, and control passes to the next instruction.

			PC	Instruction code	ACC	B	CY
	MOV	#003H, ACC	0F17H	230003H	03H	-	-
	MOV	#0F2H, B	0F1AH	2302F2H	03H	F2H	-
	BE	B, LA	0F1DH	33023FH	03H	F2H	1
	DEC	ACC	0F20H	7300H	02H	F2H	1
LA:	INC	ACC					

## BE @Rj, \_i8, r8

### Compare immediate data to indirect byte and branch near relative address if equal

Instruction code	0 0 1 1 0 1j1j0 i7i6i5i4i3i2i1i0 r7r6r5r4r3r2r1r0 (34H to 37H)
Byte count	3
Cycles	2
Function	(PC) ←(PC) + 3, if ((Rj)) = #i8 then (PC) ←(PC) + r8 j = 0, 1, 2, 3
Flags affected	CY
Interrupts enabled	Yes on the 2nd cycle

### Description

Increments the program counter (PC) three times, then compares the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0 with immediate data (i7 to i0), and if the values are equal adds the value of r7 to r0 to PC, leaving the result in PC.

If the values are different, continues to the next instruction.

If the value in the RAM address or SFR specified by the indirect address register specified by j1 to j0 is less than the immediate data (i7 to i0), the carry flag is set; if equal or more, the carry flag is cleared.

@Rj < #i8 → CY = 1

@Rj ≥ #i8 → CY = 0

### Example 1

When the BE instruction is executed, B=05H, so CY is cleared, and control branches to the label LA.

			PC	Instruction code	B	RAM	CY
						03H	
	MOV	#005H, B	0F17H	230205H	05H	-	-
	MOV	#002H, 003H	0F1AH	220302H	05H	02H	-
	BE	@R3, #5H, LA	0F1DH	37053FH	05H	02H	0
LA:	INC	B	0F5FH	6302H	06H	02H	0

### Example 2

When the BE instruction is executed, ACC\_09H, so CY is set, and control passes to the next instruction.

			PC	Instruction code	ACC	RAM	CY
						02H	
	MOV	#003H, ACC	0F17H	230003H	03H	-	-
	MOV	#000H, 002H	0F1AH	220200H	03H	00H	-
	BE	@R2, #9H, LA	0F1DH	36093FH	03H	00H	1
	DEC	ACC	0F20H	7300H	02H	00H	1
LA:	INC	ACC					



**BNE\_i8, r8**

**Compare immediate data to accumulator and branch near relative address if not equal**

```

Instruction code    0 1 0 0 0 0 0 1 i7i6i5i4i3i2i1i0 r7r6r5r4r3r2r1r0 (41H)
Byte count        3
Cycles            2
Function          (PC) ← (PC) + 3, if (ACC) π #i8 then (PC) ← (PC) + r8
Flags affected    CY
Interrupts enabled Yes on the 2nd cycle
    
```

**Description**

Increments the program counter (PC) three times, then compares immediate data (i7 to i0) with the contents of the accumulator, and if the values are unequal adds the value of r7 to r0 to PC, leaving the result in PC. If the values are equal, continues to the next instruction. If the value in the accumulator is less than the immediate data value, the carry flag is set; if equal or more, the carry flag is cleared.

```

ACC < #i8 → CY = 1
ACC ≥ #i8 → CY = 0
    
```

**Example 1**

When the BNE instruction is executed, ACC\_00H, so CY is cleared, and control branches to the label LA.

			PC	Instruction code	ACC	CY
	MOV	#002H, ACC	0F1AH	230002H	02H	-
	BNE	#000H, LA	0F1DH	41003FH	02H	0
LA:	INC	ACC	0F5FH	6300H	03H	0

**Example 2**

When the BNE instruction is executed, ACC=03H, so CY is cleared, and control passes to the next instruction.

			PC	Instruction code	ACC	CY
	MOV	#003H, ACC	0F1AH	230003H	03H	-
	BNE	#003H, LA	0F1DH	41033FH	03H	0
	DEC	ACC	0F20H	7300H	02H	0
LA:	INC	ACC				

## BNE d9, r8

### Compare direct byte to accumulator and branch near relative address if not equal

Instruction code	0 1 0 0 0 1d8 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 (42H to 43H)
Byte count	3
Cycles	2
Function	$(PC) \leftarrow (PC) + 3$ , if $(ACC) \neq (d9)$ then $(PC) \leftarrow (PC) + r8$
Flags affected	CY
Interrupts enabled	Yes on the 2nd cycle

### Description

Increments the program counter (PC) three times, then compares the contents of the RAM address or SFR specified by d8 to d0 with the contents of the accumulator, and if the values are unequal adds the value of r7 to r0 to PC, leaving the result in PC.

If the values are equal, continues to the next instruction.

If the value in the accumulator is less than the contents of the RAM address or SFR specified by d8 to d0, the carry flag is set; if equal or more, the carry flag is cleared.

$ACC < d9(\text{RAM or SFR}) \rightarrow CY = 1$

$ACC \geq d9(\text{RAM or SFR}) \rightarrow CY = 0$

### Example 1

When the BNE instruction is executed, ACC\_B, so CY is set, and control branches to the label LA.

			PC	Instruction code	ACC	B	CY
	MOV	#002H,ACC	0F17H	230002H	02H	-	-
	MON	#003H,B	0F1AH	230203H	02H	03H	-
	BNE	B,LA	0F1DH	43023FH	02H	03H	1
LA:	INC	ACC	0F5FH	6300H	03H	03H	1

### Example 2

When the BNE instruction is executed, ACC=B, so CY is cleared, and control passes to the next instruction.

			PC	Instruction code	ACC	B	CY
	MOV	#0F2H,ACC	0F17H	2300F2H	F2H	-	-
	MOV	#0F2H,B	0F1AH	2302F2H	F2H	F2H	-
	BNE	B,LA	0F1DH	43023FH	F2H	F2H	0
	DEC	ACC	0F20H	7300H	F1H	F2H	0
LA:	INC	ACC					

**BNE @Rj, \_i8, r8**

**Compare immediate data to indirect byte and branch near relative address if not equal**

Instruction code	0 1 0 0 0 1j1j0 i7i6i5i4i3i2i1i0 r7r6r5r4r3r2r1r0 (44H to 47H)
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, if ((Rj) π #i8 then (PC) " (PC) + r8 j = 0, 1, 2, 3
Flags affected	CY
Interrupts enabled	Yes on the 2nd cycle

**Description**

Increments the program counter (PC) three times, then compares the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0 with immediate data (i7 to i0), and if the values are unequal adds the value of r7 to r0 to PC, leaving the result in PC.

If the values are different, continues to the next instruction.

If the value in the RAM address or SFR specified by the indirect address register specified by j1 to j0 is less than the immediate data (i7 to i0), the carry flag is set; if equal or more, the carry flag is cleared.

@Rj < #i8 → CY = 1

@Rj ≥ #i8 → CY = 0

**Example 1**

When the BNE instruction is executed, B\_08H, so CY is set, and control branches to the label LA.

			PC	Instruction	ACC	B	CY
				code			code
	MOV	#002H, ACC	0F17H	230002H	02H	-	-
	MON	#003H, B	0F1AH	230203H	02H	03H	-
	BNE	B, LA	0F1DH	43023FH	02H	03H	1
LA:	INC	ACC	0F5FH	6300H	03H	03H	1

**Example 2**

When the BNE instruction is executed, ACC=03H, so CY is cleared, and control passes to the next instruction.

			PC	Instruction	ACC	RAM	CY
				code			
	MOV	#003H, ACC	0F17H	230003H	03H	-	-
	MOV	#000H, 002H	0F1AH	220200H	03H	00H	-
	BNE	@R2, #3H, LA	0F1DH	46033FH	03H	00H	0
	DEC	ACC	0F20H	7300H	02H	00H	0
LA:	INC	ACC					

## Subroutine Instructions

### CALL a12

#### Near absolute subroutine call

Instruction code      0 0 0a11 1a10a9a8 a7a6a5a4a3a2a1a0 (08H to 0FH, 18H to 1FH)  
 Byte count                      2  
 Cycles                              2  
 Function                            (PC) ← (PC) + 2, (SP) ← (SP) + 1, ((SP)) ← (PC7 to 0),  
     (SP) ← (SP) + 1, ((SP)) ← (PC15 to 8), (PC11 to 0) ← a12  
 Flags affected  
 Interrupts enabled                Yes on the 2nd cycle

#### Description

Increments the program counter (PC) twice, then increments the stack pointer (SP), stores the lower byte of PC at the RAM address indicated by SP; increments the stack pointer (SP) again, and stores the upper byte of PC at the RAM address indicated by SP; finally, transfers the value of a11 to a0 to bits 11 to 00 of PC.

#### Example 1

The value of label LA is 0F0EH.

		PC	Instruction code	SP	RAM	RAM	
					20H	21H	
	MOV	#01FH,SP	0FFAH	23061FH	1FH	-	-
	CALL	LA	0FFDH	1F0EH	21H	FFH	0FH
LA:	INC	ACC	0F0EH	6300H	21H	FFH	0FH
	RET		0F10H	A0H		1FH	FFH 0FH
	NOP		0FFFH	00H		1FH	FFH 0FH

#### Example 2

The value of label LA is 1F0EH.

		PC	Instruction code	SP	RAM	RAM		
					20H	21H		
	MOV	#01FH,SP	0FFBH	23061FH	1FH	-	-	
	CALL	LA	0FFEh	1F0EH	21H	00H	10H	
LA:	INC	ACC	1F0EH	6300H	21H	00H	10H	
	RET		1F10H	A0H		1FH	00H	10H
	INC	ACC	1000H	6300H		1FH	00H	10H

**CALLF a16**

**Far absolute subroutine call**

Instruction code     0 0 1 0 0 0 0 0 a15a14a13a12a11a10a9a8 a7a6a5a4a3a2a1a0 (20H)  
 Byte count             3  
 Cycles                 2  
 Function               (PC) ← (PC) + 3, (SP) ← (SP) + 1, ((SP)) ← (PC7 to 0),  
                           (SP) ← (SP) + 1, ((SP)) ← (PC15 to 8), (PC) ← a16  
 Flags affected  
 Interrupts enabled     Yes on the 2nd cycle

**Description**

Increments the program counter (PC) three times, then increments the stack pointer (SP), stores the lower byte of PC at the RAM address indicated by SP; increments the stack pointer (SP) again, and stores the upper byte of PC at the RAM address indicated by SP; finally, transfers the value of a15 to a0 to bits 15 to 00 of PC.

**Example 1**

The value of label LA is 0F0EH.

			PC	Instruction code	SP	RAM	RAM
						20H	21H
	MOV	#01FH,SP	0FF9H	23061FH	1FH	-	-
	CALLF	LA	0FFCH	200F0EH	21H	FFH	0FH
LA:	INC	ACC	0F0EH	6300H	21H	FFH	0FH
	RET		0F10H	A0H	1FH	FFH	0FH
	NOP		0FFFH	00H	1FH	FFH	0FH

**Example 2**

The value of label LA is 0F0EH.

			PC	Instruction code	SP	RAM	RAM
						20H	21H
	MOV	#01FH,SP	0FFAH	23061FH	1FH	-	-
	CALLF	LA	0FFDH	200F0EH	21H	00H	10H
LA:	INC	ACC	0F0EH	6300H	21H	00H	10H
	RET		0F10H	A0H	1FH	00H	10H
	INC	ACC	1000H	6300H	1FH	00H	10H

## CALLR r16

### Far relative subroutine call

Instruction code    0 0 0 1 0 0 0 0 r7r6r5r4r3r2r1r0 r15r14r13r12r11r10r9r8 (10H)  
 Byte count                3  
 Cycles                    4  
 Function                (PC) ← (PC) + 3, (SP) ← (SP) + 1, ((SP)) ← (PC7 to 0),  
                           (SP) ← (SP) + 1, ((SP)) ← (PC15 to 8), (PC) ← (PC)  
                           - 1 + r16  
 Flags affected  
 Interrupts enabled        Yes on the 4th cycle

### Description

Increments the program counter (PC) three times, then increments the stack pointer (SP), stores the lower byte of PC at the RAM address indicated by SP; increments the stack pointer (SP) again, and stores the upper byte of PC at the RAM address indicated by SP; finally, decrements PC, then adds the value of r15 to r0 to PC, leaving the result in PC.

### Example 1

The value of label LA is 1100H.

			PC	Instruction code	SP	RAM	RAM
						20H	21H
	MOV	#01FH,SP	0FF9H	23061FH	1FH	-	-
	CALLR	LA	0FFCH	100201H	21H	FFH	0FH
LA:	INC	ACC	1100H	6300H	21H	FFH	0FH
	RET		1102H	A0H	1FH	FFH	0FH
	NOP		0FFFH	00H	1FH	FFH	0FH

### Example 2

The value of label LA is 1100H.

			PC	Instruction code	SP	RAM	RAM
						20H	21H
	MOV	#01FH,SP	0FFCH	23061FH	1FH	-	-
	CALLR	LA	0FFDH	100101H	21H	00H	10H
LA:	INC	ACC	1100H	6300H	21H	00H	10H
	RET		1102H	A0H	1FH	00H	10H
	INC	ACC	1000H	6300H	1FH	00H	10H

**RET**

**Return from subroutine**

Instruction code    1 0 1 0 0 0 0 0 (A0H)  
 Byte count            1  
 Cycles                2  
 Function              (PC15 to 8) ← ((SP)), (SP) ← (SP) - 1, (PC7 to 0) ← ((SP)), (SP) ← (SP) - 1  
 Flags affected  
 Interrupts enabled    Yes on the 2nd cycle

**Description**

Transfers the contents of RAM indicated by the stack pointer (SP) to the upper byte of the program counter (PC), then decrements SP, then transfers the contents of RAM indicated by the stack pointer (SP) to the lower byte of the program counter (PC), then decrements SP again.

**Example 1**

The value of label LA is 0F0EH.

			PC	Instruction code	SP	RAM	RAM
						20H	21H
	MOV	#01FH,SP	0FF9H	23061FH	1FH	-	-
	CALLF	LA	0FFCH	200F0EH	21H	FFH	0FH
LA:	INC	ACC	0F0EH	6300H	21H	FFH	0FH
	RET		0F10H	A0H	1FH	FFH	0FH
	NOP		0FFFH	00H	1FH	FFH	0FH

**Example 2**

The value of label LA is 0F0EH.

			PC	Instruction code	SP	RAM	RAM
						20H	21H
	MOV	#01FH,SP	0FFAH	23061FH	1FH	-	-
	CALLF	LA	0FFDH	200F0EH	21H	00H	10H
LA:	INC	ACC	0F0EH	6300H	21H	00H	10H
RET			0F10H	A0H	1FH	00H	10H
	INC	ACC	1000H	6300H	1FH	00H	10H

## RETI

### Return for interrupt

Instruction code	1 0 1 1 0 0 0 0 (B0H)
Byte count	1
Cycles	2
Function	(PC15 to 8) $\leftarrow$ ((SP)), (SP) $\leftarrow$ (SP) - 1, (PC7 to 0) $\leftarrow$ ((SP)), (SP) $\leftarrow$ (SP) - 1
Flags affected	
Interrupts enabled	No

### Description

Transfers the contents of RAM indicated by the stack pointer (SP) to the upper byte of the program counter (PC), then decrements SP, then transfers the contents of RAM indicated by the stack pointer (SP) to the lower byte of the program counter (PC), then decrements SP again, and resumes the interrupt handling function which was inhibited while handling an interrupt.

### Example 1

		PC	Instruction code
NOP		0FFAH	00H
NOP		0FFBH	00H
MOV	#001H,ACC	0FFCH	23001H $\leftarrow$ external interrupt 0 occurs
INC	ACC	0003H	6300H
RETI		0005H	B0H
NOP		0FFFH	00H

### Example 2

		PC	Instruction code
NOP		0FFCH	00H
MOV	#00EH,B	0FFDH	23020EH $\leftarrow$ external interrupt 1 occurs
INC	ACC	0013H	6300H
RETI		0015H	B0H
INC	ACC	1000H	6300H



## Bit Manipulation Instructions

### CLR1 d9, b3

#### Clear direct bit

Instruction code	1 1 0d8 1b2b1b0 d7d6d5d4d3d2d1d0 (C8H to CFH, D8H to DFH)
Byte count	2
Cycles	1
Function	(d9, b3) ← 0
Flags affected	
Interrupts enabled	Yes

#### Description

Clears the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0.

#### Example 1

		ACC		
MOV	#001H,ACC	01H	0000	0001B
CLR1	ACC,0	00H	0000	0000B

#### Example 2

			RAM	
			7FH	
MOV	#001H,07FH	01H	0000	0001B
CLR1	07FH,0	00H	0000	0000B

---

**Caution:** When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

---

## SET1 d9, b3

### Set direct bit

Instruction code	1 1 1d8 1b2b1b0 d7d6d5d4d3d2d1d0	(E8H to EFH, F8H to FFH)
Byte count	2	
Cycles	1	
Function	(d9, b3) ← 1	
Flags affected		
Interrupts enabled	Yes	

### Description

Set the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0.

### Example 1

		ACC		
MOV	#000H, ACC	00H	0000	0000B
SET1	ACC, 7	80H	1000	0000B

### Example 2

		RAM		
		7FH		
MOV	#001H, 07FH	01H	0000	0001B
SET1	07FH, 6	41H	0100	0001B

---

**Caution:** When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

---

**NOT1 d9, b3****Not direct bit**

Instruction code	1 0 1d8 1b2b1b0 d7d6d5d4d3d2d1d0 (A8H to AFH, B8H to BFH)
Byte count	2
Cycles	1
Function	(d9, b3) ← (d9, b3)
Flags affected	
Interrupts enabled	Yes

**Description**

Inverts the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0.

**Example 1**

		ACC		
MOV	#000H, ACC	00H	0000	0000B
NOT1	ACC, 7	80H	1000	0000B
NOT1	ACC, 7	00H	0000	0000B

**Example 2**

		RAM		
		7FH		
MOV	#001H, 07FH	01H	0000	0001B
NOT1	07FH, 6	41H	0100	0001B
NOT1	07FH, 6	01H	0000	0001B

---

**Caution:** When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

---

## Miscellaneous Instruction

### **NOP**

**No operation**

Instruction code	0 0 0 0 0 0 0 0 (00H)
Byte count	1
Cycles	1
Function	
Flags affected	
Interrupts enabled	Yes

### **Description**

Consumes one clock cycle.

## Macro Instruction

**CHANGE <label (or address)>**

**Change program mode**

### Description

Switches between system BIOS in ROM and user program in flash memory.

1) Executed while running program in ROM:

- Switch from system BIOS to application
- Set the program counter to the address in flash memory specified by the label or address.

---

**Caution:** From the system BIOS, flash memory bank 0 address 0000H is accessed. This address is fixed.

---

2) Executed while running an application:

- Switch from application to system BIOS (when LDCEXT=0)
- Set the program counter to the program address in ROM specified by the label or address.

---

**Caution:** When the CHANGE instruction is executed with LDCEXT=1 and while an application is running, a jump to the system BIOS is not carried out. In this case, there is a jump to the address in flash memory specified by the CHANGE instruction.

---

3) The program mode switch occurs after the dedicated macro instruction has executed.

4) Interrupts are disabled.

---

**Caution:** Install the GHEAD.ASM file included with the Visual Memory development set assembler, to call the system BIOS. This calls OS functions without needing to be aware of ROM addresses.

---

---

**Reference:** For details of operating system function calls, refer to the "System BIOS" section of the Visual Memory Hardware Manual.

---



# ***LC86K Instruction Set Summary***

---

**Caution:** An asterisk in the "Mnemonic" column for an instruction indicates that in byte or bit addressing the port latch is selected. For these instructions, an external signal supplied to the port is selected.

---





# **Assembler Pseudoinstructions**

A pseudoinstruction differs from an ordinary instruction (such as ADD or MOV in the LC86K instruction set); it gives directives or definitions to the assembler, and a pseudoinstruction of itself does not generate a machine instruction. (This does not apply to JMPO and other optimization pseudoinstructions, or to the CHANGE pseudoinstruction.) Pseudoinstructions are often used in combination with ordinary instructions.

<b>Group</b>	<b>Pseudoinstruction</b>	<b>Function</b>
Linking control	ORG WORLD CSEG DSEG END PUBLIC EXTERN OTHER_SIDE_SYMBOL	Specify origin Select the ROM to hold code Declare the beginning of a code segment Declare the beginning of a data segment End program Declare public symbol Declare external symbol Declare CHANGE instruction jump label
Symbol definitions	EQU SET	Assign a fixed value Assign temporary value
Data definitions	DB DW DC DS	Define byte data Define word data Define character string data Define byte area
Macro control	MACRO REPT IRP IRPC ENDM EXITM LOCAL	Define macro Repeat macro Iteration macro Character string macro End macro definition End macro expansion Define local label

Conditional assembly	IFDEF IFNDEF IFB IFNB IFE IFNE IFIDN IFDIF ELSE ENDIF .PRINTX .LIST .XLIST .MACRO .XMACRO .IF .XIF	Assemble if defined Assemble if undefined Assemble if operand empty Assemble if operand nonempty Assemble if zero Assemble if nonzero Assemble if identical Assemble if different Else case of conditional assembly End conditional assembly Display message during assembly Resume listing Suppress listing List macro expansions End macro expansion listing List skipped statements in conditional assembly End listing of skipped statements Assemble if operand empty
Miscellaneous	INCLUDE TITLE PAGE CHIP COMMENT WIDTH BANK CHANGE RADIX	Include file Set listing title New page Specify chip for assembly Add comment to object file Specify columns in listing file Specify RAM bank Jump between flash memory and ROM Specify default radix

Optimization	JMPO BRO CALLO BZO  BNZO  BPO  BPCO  BNO  DBNZO  BEO  BNEO	Optimized JMP instruction Optimized BR instruction Optimized CALL instruction BZ instruction guaranteeing no address error BNZ instruction guaranteeing no address error BP instruction guaranteeing no address error BPC instruction guaranteeing no address error BN instruction guaranteeing no address error DBNZ instruction guaranteeing no address error BE instruction guaranteeing no address error BNE instruction guaranteeing no address error Optimized BR instruction Optimized CALL instruction BZ instruction guaranteeing no address error BNZ instruction guaranteeing no address error BP instruction guaranteeing no address error BPC instruction guaranteeing no address error BN instruction guaranteeing no address error DBNZ instruction guaranteeing no address error
--------------	---	--

### ORG

**Specify origin**

#### Syntax

**ORG** expression

#### Description

The ORG pseudoinstruction specifies the start address in program memory (flash memory) as expression. Expression must be a numeric constant, or an expression which can be evaluated at assembly time.

#### Example

### WORLD

**Select the ROM to hold code**

#### Syntax

**WORLD** selection

#### Description

This specifies the ROM which will hold the assembled code. This pseudoinstruction is only valid when the target chip is the LC86800 series. There are three values which can be specified for selection.

INTERNAL	Store in the on-chip ROM.
EXTERNAL	Store in flash memory bank 0.
EXTERNAL_DATA	Store in flash memory bank 1.

---

**Caution:** For Visual Memory, always specify EXTERNAL. Other specifications may lead to data corruption or misoperation.

---

If there is more than one WORLD pseudoinstruction in a single file, an error results. For chips other than the chips other than the LC86800 series, if a value other than INTERNAL is selected for the WORLD pseudoinstruction, an error results.

### **CSEG**

#### **Declare the beginning of a code segment**

#### **Syntax**

#### **CSEG mode**

#### **Description**

This indicates to the assembler the beginning of a segment holding program code. When mode is not specified or is INBLOCK, the segment is aligned within 4K boundaries. If the mode is FREE, this indicates that the segment can be located regardless of 4K boundaries.

#### **Example**

### **DSEG**

#### **Declare the beginning of a data segment**

#### **Syntax**

#### **DSEG**

#### **Description**

This indicates to the assembler the beginning of a segment holding data.

---

**Caution:** Data segments are copied into RAM. It is not possible to open a data segment in flash memory.

---

### **Example**

**END**  
**End program**

### **Syntax**

**END**

### **Description**

This indicates the end of the source program. When the assembler encounters this instruction, it ends the pass currently being executed, so any text beyond this point, even if valid statements, is ignored.

### **Example**

**PUBLIC**  
**Declare public symbol**

### **Syntax**

**PUBLIC**    **symbol {, symbol}**

### **Description**

The PUBLIC pseudoinstruction declares that a symbol defined in the program can be referenced from other source files.

### **Example**

---

**Caution:**    To reference a symbol defined in another source file, it must be declared EXTERN.  
                  To allow a symbol in this file to be referenced from another file, it must be declared PUBLIC.

---

```
page:1 <public.ASM>
ERR          SEQ.          S LOC. OBJ.  SOURCE STATEMENTS
0001          0001          ; sample program for PUBLIC
0002          chip          lc866032
0003          public        label1, label2
0004
0005          cseg          inblock
0006          C 0000 220000  label1:  mov  #00, data1
0007          C 0003 23033C          mov  #60, c
0008          C 0006 A0          ret
0009
0010          C 0007 6200          label2:  inc  data1
0011          C 0009 0200          ld  data1
0012          C 000B 410A05          bne  #10, label3
0013          C 000E 220000          mov  #00, data1
0014          C 0011 6201          inc  data2
0015
0016          C 0013 7303          label3:  dec  c
0017          C 0015 A0          ret
0018
0019          dseg
0020          D 0000          data1:  ds  1
0021          D 0001          data2:  ds  1
0022
0023          end
```

---

**Note:** The combination of PUBLIC and EXTERN declarations allows a symbol to be referenced even when it is defined in another file.

---

### EXTERN

#### Declare external symbol

#### Syntax

**EXTERN [segment: ] symbol {, [segment: ] symbol}**

#### Description

The EXTERN pseudoinstruction is used when a symbol or symbols are defined in other source program files. The optional segment parameter is either CSEG or DSEG, indicating the segment type. If this is not specified, a code segment, CSEG, is the default.

---

**Reference:** For examples see the previous item "PUBLIC - Declare public symbol."

---

### OTHER\_SIDE\_SYMBOL

#### Declare CHANGE instruction jump label

### **Syntax**

**OTHER\_SIDE\_SYMBOL**label {, label}

### **Description**

This declares an address label which can be specified as the operand of a CHANGE instruction, which in the LC86800 series is used for switching between ROM and flash memory. The label declared is a type of external symbol, but one difference is that in a source file of code to be stored in ROM, a label is declared in flash memory (or in ROM in a source file of code to be stored in flash memory). This pseudoinstruction is only valid for the LC86800 series, and in other cases an error results.

---

**Reference:** For examples, see under "CHANGE - Jump between flash memory and ROM in this chapter."

---

### **EQU**

#### **Assign a fixed value**

### **Syntax**

**Symbolname EQU expression**

### **Description**

The EQU pseudoinstruction assigns the value expression to symbolname. A symbol defined with the EQU pseudoinstruction cannot be redefined. Used appropriately, the EQU pseudoinstruction can aid the visual identification of constant data, and improve maintenance efficiency.

### **Example**

### **SET**

#### **Assign temporary value**

### **Syntax**

**Symbolname SET expression**

### **Description**

The SET pseudoinstruction assigns the value expression to symbolname. A symbol defined with the SET pseudoinstruction can be redefined by a subsequent SET. However, a symbol set with this pseudoinstruction cannot be the subject of a PUBLIC declaration, nor can it be redefined with EQU.

### Example

#### DB

#### Define byte data

#### Syntax

**Labelname DB expression {, expression}**

#### Description

The DB pseudoinstruction stores the 8-bit data value or values represented by expression in program memory (ROM). Any number of operands may be specified, separated by commas. When two or more operands are specified, they are evaluated in order left to right, and stored in successive addresses. If there are two commas with nothing between them, this is interpreted as a zero value.

### Example

In the above example, because the "db area" statement references the symbol "area," which is a 16-bit value, at assembly time a warning level message, "Value is out of range," is generated. The bottom eight bits of the value are used in the object code.

#### DW

#### Define word data

#### Syntax

**labelname DW expression {, expression}**

#### Description

The DW pseudoinstruction stores the 16-bit data value or values represented by expression in program memory (ROM). The more significant byte is stored first, and the less significant byte at the address one higher. Any number of operands may be specified, separated by commas. When two or more operands are specified, they are stored in successive addresses. If there are two commas with nothing between them, this is interpreted as a zero value.

### Example

#### DC

#### Define character string data

#### Syntax

**labelname DC "string"**



### Description

This stores the ASCII codes of string (a character string constant) in sequence in program memory (ROM).

---

**Reference:** For details of character string constants, see Section 20.7, "Character String Constants."

---

### Example

**DS**

**Define byte area**

### Syntax

**labelname DS absolute\_expression**

### Description

The DS pseudoinstruction allocates any area of data memory (RAM) of the number of bytes specified by `absolute_expression`. The `absolute_expression` must have a value completely determined at assembly time. This pseudoinstruction can only be used after a DSEG pseudoinstruction.

---

**Caution:** A DS pseudoinstruction can only be used to allocate RAM (a data segment). It cannot be used for flash memory. Use DB or DW statements instead.

---

### Example

**MACRO**

**Define macro**

### Syntax

**name      MACRO parameter {, parameter}**

### Description

The MACRO pseudoinstruction defines a macro. The statements from the MACRO pseudoinstruction to the following ENDM pseudoinstruction form the body of the macro. The parameter name is the name by which the macro can be called, which is replaced by the body of the macro, and is therefore mandatory. The formal parameter list specified by parameter is optional, depending on the macro being defined.

---

**Caution:** When calling one macro from within another, or when using a pseudoinstruction such as IFB which requires angle brackets (<>), a sufficient depth of angle brackets to correspond to the nesting level is required. See the section "EXITM – End macro expansion" in this chapter.

---

### **Example**

#### **REPT**

##### **Repeat macro**

### **Syntax**

#### **REPT count**

### **Description**

The REPT pseudoinstruction repeats the statements up to the ENDM instruction, generating the number of copies specified by count. This value can be any integer from 1 to 65535.

### **Example**

In the following example, the area not occupied by the program is filled with NOP codes (for a 256-byte boundary).

#### **IRP**

##### **Iteration macro**

### **Syntax**

#### **IRP parameter, argument {, argument}...**

### **Description**

The IRP pseudoinstruction repeats the statements up to the ENDM instruction, generating one copy for each argument specified. In each copy, parameter is replaced by the corresponding argument.

### **Example**

#### **IRPC**

##### **Character string macro**

### **Syntax**

#### **IRPC parameter, string**

### **Description**

The IRPC pseudoinstruction repeats the statements up to the ENDM instruction, generating one copy for each character in string. As distinct from a character string constant, string is not enclosed in quotation marks. Codes beginning with a backslash cannot be used. In each copy, parameter is replaced by the corresponding character in string.

### **Example**

#### **ENDM**

**End macro definition**

### **Syntax**

#### **ENDM**

### **Description**

This marks the end of a macro definition.

### **Example**

#### **EXITM**

**End macro expansion**

### **Syntax**

#### **EXITM**

### **Description**

The EXITM pseudoinstruction ends expansion of a macro. In combination with conditional assembly pseudoinstructions, this can be used to create different forms of expansion of a macro depending on the arguments supplied.

### **Example**

#### **LOCAL**

**Define local label**

### **Syntax**

**LOCAL name {, name}**

### **Description**

The LOCAL pseudoinstruction declares a label which can be used internally to the body of the macro. During macro expansion, the name declared in the LOCAL pseudoinstruction is replaced by the assembler with a unique identifier to avoid name conflicts.

### Example

```
; sample program for LOCAL

b_ne          macro          chip 1c864008
               local        val,dst
               skip
               be    val,skip
               bro   dst

skip:

cseg

org           b_ne #0, over
200h
over:        nop
under:      nop

end
```

In the above example, the BRO pseudoinstruction is used to define the B\_NE macro which generates different instructions depending on the destination of a jump; this is then used in the example. The following is the result of assembly.

### IFDEF

#### Assemble if defined

#### Syntax

**IFDEF** symbol

#### Description

If symbol is defined, the statements after the IFDEF pseudoinstruction until the next ELSE or ENDIF are assembled.

#### Example

### IFNDEF

#### Assemble if undefined

#### Syntax

**IFNDEF** symbol

#### Description

If symbol is undefined, the statements after the IFNDEF pseudoinstruction until the next ELSE or ENDIF are assembled.

### **Example**

#### **IFB**

**Assemble if operand empty**

#### **Syntax**

**IFB <argument>**

#### **Description**

If argument is empty, the statements after the IFB pseudoinstruction until the next ELSE or ENDIF are assembled. "Empty" means that there are no characters at all (even spaces or tabs) between the angle brackets in which argument must be enclosed.

### **Example**

#### **IFNB**

**Assemble if operand nonempty**

#### **Syntax**

**IFNB <argument>**

#### **Description**

If argument is nonempty, the statements after the IFNB pseudoinstruction until the next ELSE or ENDIF are assembled. "Empty" means that there are no characters at all (even spaces or tabs) between the angle brackets in which argument must be enclosed.

### **Example**

#### **IFE**

**Assemble if zero**

#### **Syntax**

**IFE expression**

#### **Description**

If the value of expression is zero, the statements after the IFE pseudoinstruction until the next ELSE or ENDIF are assembled.

### **Example**

#### **IFNE**

**Assemble if nonzero**

#### **Syntax**

**IFNE expression**

#### **Description**

If the value of expression is nonzero, the statements after the IFNE pseudoinstruction until the next ELSE or ENDIF are assembled.

### **Example**

#### **IFIDN**

**Assemble if identical**

#### **Syntax**

**IFIDN <string1>, <string2>**

#### **Description**

If the two strings string1 and string2 are identical, the statements after the IFIDN pseudoinstruction until the next ELSE or ENDIF are assembled. The strings must be enclosed in angle brackets, and within them, comparison is carried out with spaces and tabs considered significant.

### **Example**

#### **IFDIF**

**Assemble if different**

#### **Syntax**

**IFDIF <string1>, <string2>**

#### **Description**

If the two strings string1 and string2 are different, the statements after the IFDIF pseudoinstruction until the next ELSE or ENDIF are assembled. The strings must be enclosed in angle brackets, and within them, comparison is carried out with spaces and tabs considered significant.

### **Example**

#### **ELSE**

**Else case of conditional assembly**

#### **Syntax**

#### **ELSE**

#### **Description**

The statements after the ELSE pseudoinstruction until the next ENDIF are assembled when the test condition of the preceding IF pseudoinstruction fails to hold.

---

**Reference:** See under "IFDEF - Assemble if defined" in this section.

---

#### **ENDIF**

**End conditional assembly**

#### **Syntax**

#### **ENDIF**

#### **Description**

Marks the end of a conditional assembly.

---

**Reference:** See under "IFDEF - Assemble if defined" in this section.

---

#### **.PRINTX**

**Display message during assembly**

#### **Syntax**

**.PRINTX"string"**

#### **Description**

The .PRINTX pseudoinstruction displays the character string constant string during assembly.

---

**Reference:** For details of character string constants, see Section 20.7, "Character String Constants."

---

### **Example**

**.LIST**

**Resume listing**

### **Syntax**

**.LIST**

### **Description**

The .LIST pseudoinstruction resumes listing output, when it has been suppressed with the .XLIST pseudoinstruction.

### **Example**

**.XLIST**

**Suppress listing**

### **Syntax**

**.XLIST**

### **Description**

The .XLIST pseudoinstruction suppresses output to the listing file.

---

**Reference:** See under ".LIST - Resume listing," in this section.

---

**.MACRO**

**List macro expansions**

### **Syntax**

**.MACRO**

### **Description**

The .MACRO pseudoinstruction causes the expanded body of macro calls to be output to the listing file.

### **Example**



### **.XMACRO**

#### **End macro expansion listing**

#### **Syntax**

### **.XMACRO**

#### **Description**

The .XMACRO pseudoinstruction ends the output of expanded macro calls to the listing.

---

**Reference:** For an example, see under the previous item, ".MACRO - List macro expansions."

---

### **REPT**

#### **Repeat macro**

#### **Syntax**

#### **REPT count**

#### **Description**

The REPT pseudoinstruction repeats the statements up to the ENDM instruction, generating the number of copies specified by count. This value can be any integer from 1 to 65535.

#### **Example**

In the following example, the area not occupied by the program is filled with NOP codes (for a 256-byte boundary).

### **IRP**

#### **Iteration macro**

#### **Syntax**

**IRP parameter, argument {, argument}...**

### **Description**

The IRP pseudoinstruction repeats the statements up to the ENDM instruction, generating one copy for each argument specified. In each copy, parameter is replaced by the corresponding argument.

### **Example**

#### **IRPC**

#### **Character string macro**

### **Syntax**

**IRPC** parameter, string

### **Description**

The IRPC pseudoinstruction repeats the statements up to the ENDM instruction, generating one copy for each character in string. As distinct from a character string constant, string is not enclosed in quotation marks. Codes beginning with a backslash cannot be used. In each copy, parameter is replaced by the corresponding character in string.

### **Example**

#### **ENDM**

#### **End macro definition**

### **Syntax**

#### **ENDM**

### **Description**

This marks the end of a macro definition.

### **Example**

#### **EXITM**

End macro expansion

### **Syntax**

#### **EXITM**

### Description

The EXITM pseudoinstruction ends expansion of a macro. In combination with conditional assembly pseudoinstructions, this can be used to create different forms of expansion of a macro depending on the arguments supplied.

### Example

#### LOCAL

#### Define local label

### Syntax

**LOCAL** name {, name}

### Description

The LOCAL pseudoinstruction declares a label which can be used internally to the body of the macro. During macro expansion, the name declared in the LOCAL pseudoinstruction is replaced by the assembler with a unique identifier to avoid name conflicts.

### Example

```
; sample program for LOCAL
                                chip                1c864008
b_ne                            macro               val,dst
                                local               skip
                                be    val,skip
                                bro    dst
skip:
                                endm
cseg
                                b_ne    #0, over
org                             200h
over:                            b_ne            #0, under
                                nop
under:                            nop
                                end
```

In the above example, the BRO pseudoinstruction is used to define the B\_NE macro which generates different instructions depending on the destination of a jump; this is then used in the example. The following is the result of assembly.

### **IFDEF**

**Assemble if defined**

#### **Syntax**

**IFDEF** symbol

#### **Description**

If symbol is defined, the statements after the IFDEF pseudoinstruction until the next ELSE or ENDIF are assembled.

#### **Example**

### **IFNDEF**

**Assemble if undefined**

#### **Syntax**

**IFNDEF** symbol

#### **Description**

If symbol is undefined, the statements after the IFNDEF pseudoinstruction until the next ELSE or ENDIF are assembled.

#### **Example**

### **IFB**

Assemble if operand empty

#### **Syntax**

**IFB** <argument>

#### **Description**

If argument is empty, the statements after the IFB pseudoinstruction until the next ELSE or ENDIF are assembled. "Empty" means that there are no characters at all (even spaces or tabs) between the angle brackets in which argument must be enclosed.

### **Example**

#### **IFNB**

**Assemble if operand nonempty**

#### **Syntax**

**IFNB** <argument>

#### **Description**

If argument is nonempty, the statements after the IFNB pseudoinstruction until the next ELSE or ENDIF are assembled. "Empty" means that there are no characters at all (even spaces or tabs) between the angle brackets in which argument must be enclosed.

### **Example**

#### **IFE**

**Assemble if zero**

#### **Syntax**

**IFE** expression

#### **Description**

If the value of expression is zero, the statements after the IFE pseudoinstruction until the next ELSE or ENDIF are assembled.

### **Example**

#### **IFNE**

**Assemble if nonzero**

#### **Syntax**

**IFNE** expression

#### **Description**

If the value of expression is nonzero, the statements after the IFNE pseudoinstruction until the next ELSE or ENDIF are assembled.

### **Example**

#### **IFIDN**

**Assemble if identical**

### **Syntax**

**IFIDN <string1>, <string2>**

### **Description**

If the two strings string1 and string2 are identical, the statements after the IFIDN pseudoinstruction until the next ELSE or ENDIF are assembled. The strings must be enclosed in angle brackets, and within them, comparison is carried out with spaces and tabs considered significant.

### **Example**

#### **IFDIF**

**Assemble if different**

### **Syntax**

**IFDIF <string1>, <string2>**

### **Description**

If the two strings string1 and string2 are different, the statements after the IFDIF pseudoinstruction until the next ELSE or ENDIF are assembled. The strings must be enclosed in angle brackets, and within them, comparison is carried out with spaces and tabs considered significant.

### **Example**

#### **ELSE**

**Else case of conditional assembly**

### **Syntax**

#### **ELSE**

### **Description**

The statements after the ELSE pseudoinstruction until the next ENDIF are assembled when the test condition of the preceding IF pseudoinstruction fails to hold.

---

**Reference:** See under "IFDEF - Assemble if defined" in this section.

---

### **ENDIF**

**End conditional assembly**

### **Syntax**

### **ENDIF**

### **Description**

Marks the end of a conditional assembly.

---

**Reference:** See under "IFDEF - Assemble if defined" in this section.

---

### **.PRINTX**

**Display message during assembly**

### **Syntax**

**.PRINTX "string"**

### **Description**

The .PRINTX pseudoinstruction displays the character string constant string during assembly.

---

**Reference:** For details of character string constants, see Section on, "Character String Constants."

---

### **Example**

### **.LIST**

**Resume listing**

### **Syntax**

**.LIST**

### **Description**

The .LIST pseudoinstruction resumes listing output, when it has been suppressed with the .XLIST pseudoinstruction.

### **Example**

**.XLIST**

**Suppress listing**

### **Syntax**

**.XLIST**

### **Description**

The .XLIST pseudoinstruction suppresses output to the listing file.

---

**Reference:** See under ".LIST - Resume listing," in this section.

---

**.MACRO**

**List macro expansions**

### **Syntax**

**.MACRO**

### **Description**

The .MACRO pseudoinstruction causes the expanded body of macro calls to be output to the listing file.

### **Example**

**.XMACRO**

**End macro expansion listing**

### **Syntax**

**.XMACRO**

### **Description**

The .XMACRO pseudoinstruction ends the output of expanded macro calls to the listing.

---

**Reference:** For an example, see under the previous item, ".MACRO - List macro expansions."

---



**.IF**

**List skipped statements in conditional assembly**

**Syntax**

**.IF**

**Description**

The .IF pseudoinstruction causes source program statements skipped in a conditional assembly to be output to the listing file.

**Example**

**.XIF**

**End listing of skipped statements**

**Syntax**

**.XIF**

**Description**

The .XIF pseudoinstruction stops source program statements skipped in a conditional assembly from being output to the listing file.

---

**Reference:** For an example, see under the previous item, ".IF - List skipped statements in conditional assembly."

---

**INCLUDE**

**Include file**

**Syntax**

**INCLUDE filename**

**Description**

The INCLUDE pseudoinstruction causes the source file specified by filename to be read into the current point in the source program and assembled. The specification of filename must include the extension. The INCLUDE pseudoinstruction can be nested to a maximum depth of nine. Note that if an END pseudoinstruction occurs in the included file, this terminates the assembly.

### **Example**

#### **TITLE**

**Set listing title**

#### **Syntax**

**TITLE string**

#### **Description**

The TITLE pseudoinstruction specifies string as the title for the listing file. Unlike a character string constant, string is not enclosed in quotation marks. It is also not possible to include codes with the backslash (\) symbol.

### **Example**

#### **PAGE**

**New page**

#### **Syntax**

**PAGE**

#### **Description**

The PAGE pseudoinstruction forces a new page in the listing file. The page break appears immediately after this pseudoinstruction.

### **Example**

#### **CHIP**

**Specify chip for assembly**

#### **Syntax**

**CHIP chipname**

### **Description**

The CHIP pseudoinstruction informs the assembler of the chip for which assembly is to be carried out. According to the value of chipname, the assembly changes the reserved words, and carries out a memory size check. This pseudoinstruction must appear at the beginning of the source file, before any other instructions or pseudoinstructions. If this pseudoinstruction is not found, the environment variable CHIPNAME is referenced. If the chip name specified by this pseudoinstruction is different from the chip specified by the CHIPNAME environment variable, a warning level error is issued.

---

**Note:** For developing Visual Memory applications, the chip name must be set to LC868700.

---

### **COMMENT**

**Add comment to object file**

#### **Syntax**

**COMMENT comment\_string**

#### **Description**

The COMMENT pseudoinstruction adds a comment directly into the assembled object code. Unlike a character string constant, comment\_string is not enclosed in quotation marks. It is also not possible to include codes with the backslash (\) symbol. The comment is stored from byte 680 of the object file. A maximum of 255 characters can be used for the comment.

#### **Example**

### **WIDTH**

**Specify columns in listing file**

#### **Syntax**

**WIDTH number**

#### **Description**

The WIDTH pseudoinstruction specifies the number of character columns in the listing file, that is, the number of characters in each line. The parameter number may be any value from 72 to 132 inclusive, but the recommended minimum is the number of columns of the source file plus 28. Although this pseudoinstruction can appear any number of times in a single source file, normally it is specified once only at the beginning of the file. If this pseudoinstruction is not found, the default listing file has 132 columns.

### Example

#### **BANK**

Specify RAM bank

#### **Syntax**

**BANK** expression

#### **Description**

The BANK pseudoinstruction supplies the bank number for symbols defined by DS pseudoinstructions for RAM after a DSEG pseudoinstruction.

### Example

#### **CHANGE**

Jump between flash memory and ROM

#### **Syntax**

**CHANGE** symbol

#### **Description**

For the LC86800 series, this is a special jump instruction for switching between code in flash memory and code in ROM (system BIOS). The operand symbol must have been declared with the pseudoinstruction OTHER\_SIDE\_SYMBOL. Note that this pseudoinstruction is special to the LC86800 series, and in other cases an error results.

---

**Note:** For Visual Memory, use this instruction to call an operating system function.

---

---

**Reference:** For details of operating system function calls, refer to the "System BIOS" section of the Visual Memory Hardware Manual.

---

### Example

#### **RADIX**

Specify default radix

#### **Syntax**

**RADIX** expression

### Description

The RADIX pseudoinstruction specifies the radix, or base, of a numeric constant with no explicit radix indication. The value of expression must be a constant value from the set 2, 8, 10, and 16. This specification takes effect from this statement until a subsequent RADIX pseudoinstruction. If this pseudoinstruction is not present, the default radix is 10.

### Example

```
Xxx          SET    10    _   interpreted by default as 10 decimal.
              RADIX  16
Xxx          SET    10    _   interpreted as 16 decimal, because of the radix value 16.
              RADIX  2
Xxx          SET    10    _   interpreted as 2 decimal, because of the radix value 2.
```

### JMPO

#### Optimized JMP instruction

#### Syntax

#### JMPO expression

### Description

The JMPO pseudoinstruction compares expression with the current location, and if this is a jump within the same block (only the bottom 12 bits of the addresses are different) generates a JMP instruction. Otherwise, that is, if the address is in a different block, or if the address cannot be determined because for example it is an external symbol, then this generates a JMPF instruction.

### Example

### BRO

#### Optimized BR instruction

#### Syntax

#### BRO expression

### Description

BRO pseudoinstruction compares expression with the current location, and if the branch address is within the range -128 to +127 generates a BR instruction; when outside the range -128 to +127 generates a BRF instruction.

### Example

### CALLO

### Optimized CALL instruction

### Syntax

**CALLO**      **expression**

### Description

The CALLO pseudoinstruction compares expression with the current location, and if this is a call within the same block (only the bottom 12 bits of the addresses are different) generates a CALL instruction. Otherwise, that is, if the address is in a different block, or if the address cannot be determined because for example it is an external symbol, then this generates a CALLF instruction.

### Example

### BZO

### BZ instruction guaranteeing no address error

### Syntax

**BZO**    **expression**

### Description

The BZO macro generates code equivalent to the BZ instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BZO macro uses a BNZ instruction, which is the logical inverse of the BZ instruction, and a BRO instruction. Enter the branch destination for expression.

### Code generation macro

```
; *** Branch near relative address if accumulator is zero ***
bzo                                macro                r8
                                   local                _next_
                                   bnz                  _next_
                                   bro                   r8
_next_:
                                   endm
```

**BNZO****BNZ instruction guaranteeing no address error****Syntax****BNZO expression****Description**

The BNZO macro generates code equivalent to the BNZ instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BNZO macro uses a BZ instruction, which is the logical inverse of the BNZ instruction, and a BRO instruction. Enter the branch destination for expression.

**Code generation macro**

```
; *** Branch near relative address if accumulator is not zero ***
bnzo                macro                r8
                    local                _next_
                    bz                    _next_
                    bro                   r8
_next_:
                    endm
```

**BPO****BP instruction guaranteeing no address error****Syntax****BPO expression****Description**

The BPO macro generates code equivalent to the BP instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BPO macro uses a BP instruction, and BR and BRO instructions. Enter the branch destination for expression.

**Code generation macro**

```
; *** Branch near relative address if direct bit is one ***
bpo                 macro                d9,b3,r8
                    local                _next_
                    local                _true_
                    bp                    d9,b3,_true_
                    br                    _next_
_true_:             bro                   r8
_next_:
                    endm
```

### BPCO

**BPC instruction guaranteeing no address error**

#### Syntax

**BPCO** expression

#### Description

The BPCO macro generates code equivalent to the BPC instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BPCO macro uses a BPC instruction, and BR and BRO instructions. Enter the branch destination for expression.

#### Code generation macro

```
; *** Branch near relative address if direct bit is one,
;
bpco          macro          and clear ***
              local         _next_
              local         _true_
              bpc           d9,b3,_true_
              br            _next_
_true_:      bro           r8
_next_:
              endm
```

### BNO

**BN instruction guaranteeing no address error**

#### Syntax

**BNO** expression

#### Description

The BNO macro generates code equivalent to the BN instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BNO macro uses a BN instruction, and BR and BRO instructions. Enter the branch destination for expression.



**Code generation macro**

```
; *** Branch near relative address if direct bit is zero ***
bno                                macro                d9,b3,r8
                                   local                _next_
                                   local                _true_
                                   bn                    d9,b3,_true_
                                   br                    _next_
_true_:                             bro                r8
_next_:
                                   endm
```

**DBNZO****DBNZ instruction guaranteeing no address error****Syntax****DBNZOexpression****Description**

The DBNZO macro generates code equivalent to the DBNZ instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The DBNZO macro uses a DBNZ instruction, and BR and BRO instructions. The function of expression is the same as in the DBNZ instruction.

**Code generation macro**

```
; *** Decrement direct byte and branch near relative address
;                                     if direct byte is not zero ***
dbnzo                                macro                d9,r8
                                   local                _next_
                                   local                _true_
                                   dbnz                 d9,_true_
                                   br                    _next_
_true_:                             bro                r8
_next_:
                                   endm
```

**BEO****BE instruction guaranteeing no address error****Syntax**

### BEO expression

#### Description

The BEO macro generates code equivalent to the BE instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BEO macro uses a BNE instruction and BRO instruction. The function of expression is the same as in the BE instruction.

#### Code generation macro

```
; *** Compare immediate data or accumulator and branch
;
; near relative address if equal ***
beo macro arg0,arg1,arg2
    local _next_
    local _txen_
    ifb <<arg2>>
        bne arg0,_next_
        bro arg1
    else
        bne arg0,arg1,_txen_
        bro arg2
    endif
    _next_:
    _txen_:
endmacro
```

### BNEO

#### BNE instruction guaranteeing no address error

#### Syntax

### BNEO expression

#### Description

The BNEO macro generates code equivalent to the BNE instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BNEO macro uses a BE instruction and BRO instruction. The function of expression is the same as in the BNE instruction.

### Code generation macro

```
; *** Compare immediate data or accumulator and branch
;
bneo      macro      arg0,arg1,arg2
           local     _next_
           local     _txen_
           ifb      <<arg2>>
           be       arg0,_next_
           bro      arg1

_next_:

           else
           be       arg0,arg1,_txen_
           bro      arg2

_txen_:

           endif
           endm
```



***Visual Memory Unit (VMU)***  
***VMU-BIOS Specifications***



# ***Table of Contents***

<b>VMU-BIOS Specifications .....</b>	<b>VME-3</b>
Outline .....	VME-3
VMU Outline .....	VME-4
System-BIOS Outline .....	VME-4
Memory Space .....	VME-5
System BIOS Functions .....	VME-7
System BIOS Data and Memory Allocation .....	VME-8
Program Layout .....	VME-8
Subroutine Call Flow .....	VME-9
Returning From User Program to Mode Selection Screen .....	VME-11
VMU Initialization .....	VME-12
Subroutine Description .....	VME-14
Flash Memory Access Functions .....	VME-14
Clock Function .....	VME-21
Automatic low battery detection function .....	VME-22
Automatic low battery detection flag .....	VME-22





# ***VMU-BIOS Specifications***

---

## **Outline**

This document describes the System BIOS functions of the backup memory system “VMU” designed for the new-generation game machine (preliminary).

## **VMU Outline**

“VMU” stands for “Visual Memory Unit”. The VMU is a backup memory cartridge equipped with a liquid-crystal display and operation buttons.

When connected to a dedicated controller in the new-generation game machine (preliminary), the VMU serves as a file backup memory and it can also display game sub screens on its LCD.

When not connected to the new-generation game machine, the VMU can function as a stand alone unit that allows displaying and deleting stored files. Two VMU units can be connected to allow file transfer.

Another application of the VMU is as a highly portable miniature game machine, using simple application programs downloaded from the new-generation game machine to the VMU. (Such application programs are called “user programs”.)

## **System-BIOS Outline**

The functions described above are implemented by several programs that are contained in an internal ROM on the VMU. These programs are called “OS programs”. OS programs consist of subroutines which can be called by user programs. Two program types (system program and header) are used to call up subroutines. The entire system consisting of OS programs, system programs, and headers is called the “System-BIOS”.

OS program subroutines are divided into subroutines that serve mainly for accessing the flash memory and subroutines for calculating time data. Application developers can use the System-BIOS to call these subroutines in user programs. This makes it easy for application developers to use VMU functions without having to deal with detailed VMU specifications.

# Memory Space

VMU uses two types of memory space: internal memory space and external memory space.

Internal memory space consists of the internal program area and internal RAM. The external memory space is made up of flash memory.

The internal program area is 64 kilobytes and contains the OS programs and system programs. User programs can reference this area as needed. The internal program area is allocated as shown in the memory map in Figure 1.4, "VMU memory map," (For information on OS programs and system programs, please refer to Section , "System BIOS Data and Memory Allocation".)

The flash memory space is 128 kilobytes, divided into two banks of 64 kilobytes each. Bank-0 is the program area and bank-1 the data area. User programs are stored in the program area. A memory map of the flash memory is shown in Figure 1.2, "Flash memory space," (For information on the internal and external program area and BIOS usage, please see "System BIOS Functions" on page 5, and the following sections.)

The internal RAM has a memory space of 1222 bytes, divided into the following four sections: main RAM, special register area, LCD video RAM (XRAM), work RAM (VTRBF).

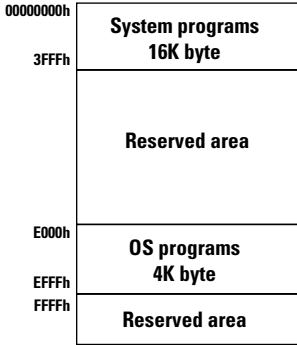
The main RAM is 512 bytes, divided into two banks of 256 bytes each. Because bank-0 is reserved for the System BIOS, user programs are generally prohibited from writing to bank-0 (except for certain cases listed in appendix 1).

The special register area is allocated to VMU specific registers (timer register, LCD control register, etc.). For information on registers and corresponding addresses, please refer to the VMU user's manual.

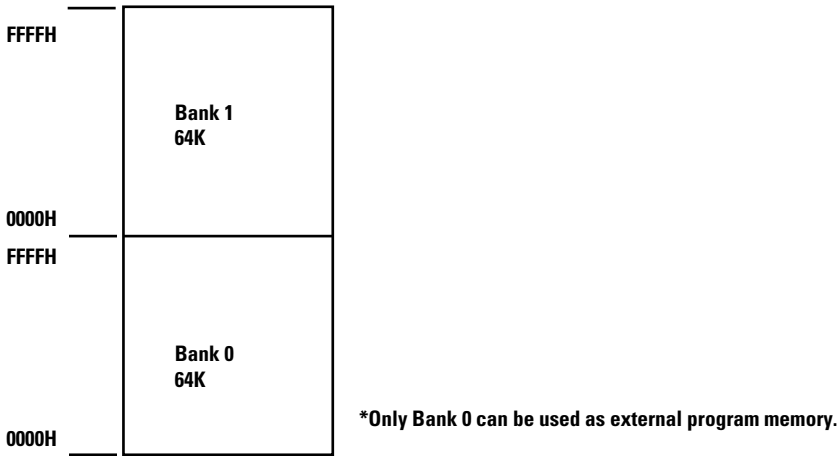
The LCD video RAM (XRAM) consists of three banks which serve for storing LCD image data. (For information on RAM usage, please refer to the VMU user's manual.)

The work RAM is 512 bytes and serves as a buffer when VMU carries out data transfer according to the Maple Bus protocol. When the VMU is operating as a standalone unit and data transfer according to the Maple Bus protocol is therefore not being carried out, user programs can use this area.

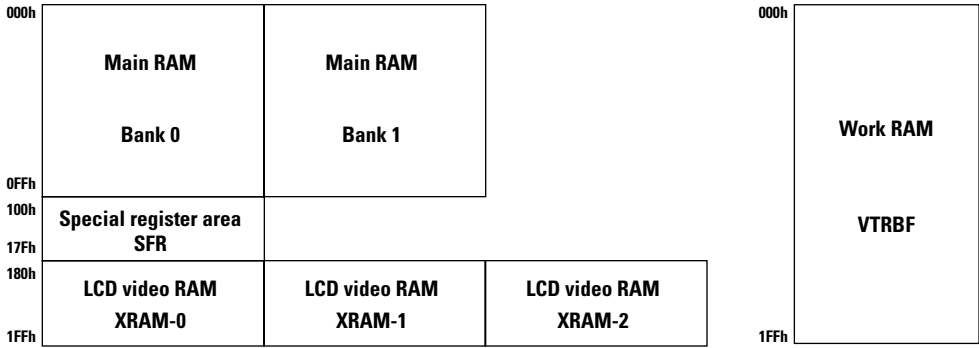
A memory map of the internal RAM is shown in Figure 1.3, "Internal RAM space," (The access procedure for the work RAM differs from normal RAM access. For information, please refer to the VMU user's manual.)



**Figure 1.1** Internal program space



**Figure 1.2** Flash memory space



\*Bank 0 of the main RAM is reserved for system programs. Except for special cases, user programs cannot use this area.

**Figure 1.3** Internal RAM space

# **System BIOS Functions**

This section explains the System BIOS functions provided for VMU.

User programs running on the VMU can access System BIOS functions by calling special subroutines. However, there are certain limitations on which System BIOS functions (subroutines) can be called by user programs.

The following functions are provided by the System BIOS.

- System initialization
  - VMU initialization function

- VMU execution mode selection

VMU comes with the following three execution modes:

- 1) Game data and user program management and editing
- 2) User program startup and return
- 3) Time display and adjustment

For details on execution mode selection, please refer to appendix 2.

- Subroutines

- Flash Memory Access Functions

- 1) Flash Memory Page Data Readout
- 2) Writing to Flash Memory
- 3) Flash Memory Verify

- Clock Function

- 1) Clock Countup Timer

For details on VMU initialization, please refer to Section , "VMU Initialization". For details on subroutines, please refer to Section , "Subroutine Description".

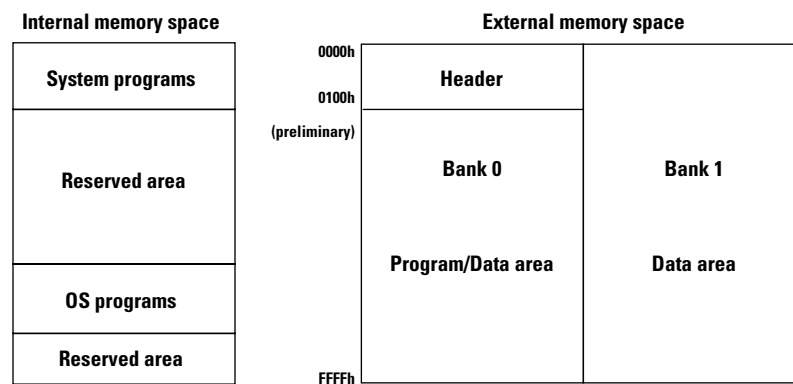
## System BIOS Data and Memory Allocation

VMU comes with certain programs for using the System BIOS functions. These programs can be classified into the following three types:

- 1) OS programs
- 2) System programs
- 3) Header

### Program Layout

The actual programs are arranged in memory as follows.



**Figure 1.4** VMU memory map

Details are explained below.

#### System programs

System programs are required for using the VMU as a memory device. Major system programs are the file management system, clock functions, and programs for data transfer according to the Maple standard. A program for calling subroutines from the external memory space (user programs) is also located here. (A flow diagram showing the call-up process of specified subroutines is shown in Section , "Subroutine Call Flow".)

The VMU initialization program is located in this area. For details on the initialization program, please refer to Section , "VMU Initialization".

## OS programs

The VMU program subroutines are located here. For information on subroutines that can be called by user programs, please refer to Section , "Subroutine Description".

The method of accessing to this area is also shown in Section , "Subroutine Call Flow".

## Header

Contains information about internal memory space processing routines and return procedures from the internal memory space. Because this area also contains interrupt vectors for internal use by user programs, its source code is being made available to application developers. It also contains information about return from user programs to the mode selection screen. User programs must use this information to return to the file management system. (For details on mode selection screen, please refer to Appendix 2.) Within the given specifications, the area content may be modified by developers.

## Subroutine Call Flow

This section explains the operation flow that occurs when a user programs calls OS program subroutines and then returns to the user program. An actual flow diagram is shown in Figure 1.5, "OS program call flow,".

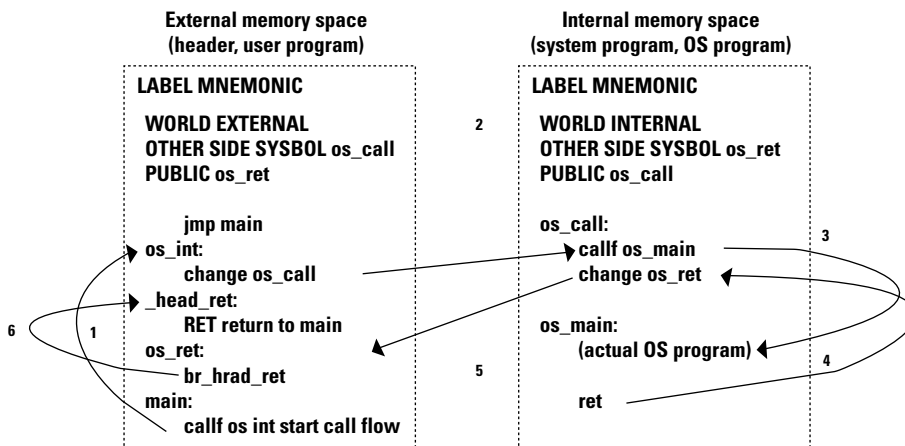


Figure 1.5 OS program call flow

## Label processing description

- external memory space
  - main: Main program in user program
  - os\_int: This subroutine shifts processing to internal memory space.  
In the example, processing passes to the internal memory space when the subroutine is called, and the main program resumes upon return from the internal memory space. This subroutine is included in the header.
  - os\_ret: Subroutine for returning to external memory space.  
The “change” command serves to return to this label from the internal memory space. After returning, processing moves to the interrupt return routine in the header.
  
- internal memory space
  - os\_call: Serves to call an OS program and return to the external memory space.  
After the OS program subroutine has executed, processing returns to the external memory space.
  - os\_main: Main OS program which executes the various subroutines.

The sample flow shown in Figure 1.5, “OS program call flow,” assumes that a user program is executing in the external memory space.

- 1) When wishing to use an OS program during execution of an external program, call the “os\_int” subroutine. Interrupt processing routines which need to jump to an OS program contain an “os\_int” subroutine.
- 2) The “change” command in the os\_int subroutine jumps to the OS program call routine (os\_call) placed in the internal program area.
- 3) The OS program call routine calls the actual OS program subroutine (os\_main). From this point on, the OS program starts to execute.
- 4) When the OS program execution is finished, the RET command jumps to the next address of the call command in the OS program call routine. In the OS program call routine, the call command is always followed by a change command which moves processing to the external program area.
- 5) After returning from the OS program subroutine, the change command passes processing over to the external program. This program contains a subroutine (os\_ret) that is called when returning from an internal program. The subroutine position is fixed. These programs are distributed to application developers as a library. Such programs are called headers. (The sample program contains the headers “os\_int” and “os\_ret”.)
- 6) From the above described external program return routine, processing returns to the “os\_int” subroutine and then by the RET command to the main program (main).

---

**Note:** Label names in the sample program are all preliminary. Label names will be different in the actual System-BIOS.

---

\* “change” command

The “change” command serves to move processing from the external memory space to the internal memory space, or from the internal memory space to external memory. By executing this command, a program that is currently executing in internal memory space (or external memory space) moves to external memory space (or internal memory space). The program counter is reset to the specified label (or address).



## Returning From User Program to Mode Selection Screen

When a user program is executing, if the user presses the MODE button on VMU, the user program will terminate immediately and processing will return to the mode selection screen.

This section explains the operation flow from user program to the mode selection screen when the MODE button is pressed while a user program is executing.

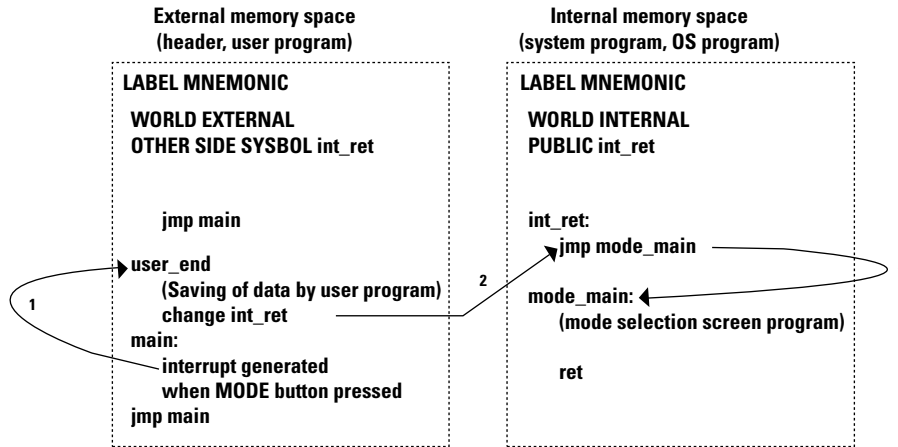


Figure 1.6 Operation flow of returning to mode selection

### Label processing description

- external memory space

main: Main program in a user program.

A user program must contain description to allow for pressing of the MODE button to jump to the OS program return subroutine.

user\_end: Subroutine to terminate a user program in execution and move processing to the OS program. If data in the executing user program needs to be saved, then be sure to include this information in the user program so that the subroutine will save it before returning to the OS program. (The OS program does not keep data.)

- internal memory space

int\_ret: Return routine to serve as entry to returning to the internal memory space when a user program terminates. When processing returns to the internal memory area, the mode selection program will start.

mode\_main: Mode selection program.

For details on mode selection specification, please refer to Appendix 2.

The sample program flow in Figure 1.6, "Operation flow of returning to mode selection," assumes the user program is executing in the external memory space.

- 1) While an external program is executing, pressing the MODE button will jump to the user\_end subroutine. In the user\_end subroutine, the "change" command will shift processing to the internal memory space. Therefore, if data in the executing user program needs to be saved, then be sure to save it before executing the "change" command.
- 2) When program control jumps from the user program to the user\_end subroutine, the "change" command inside the user\_end subroutine will shift processing to the mode\_ret subroutine in the internal memory space.
- 3) When processing moves from the external memory space to the mode\_ret subroutine, the mode selection program will start.

### \* "change" command

The "change" command serves to move processing from the external memory space to the internal memory space, or from the internal memory space to external memory. By executing this command, a program that is currently executing in the internal memory space (or external memory space) moves to the external memory space (or internal memory space). The program counter is reset to the specified label (or address).

## VMU Initialization

This section explains the initialization that is performed at VMU startup.

The VMU is automatically initialized in the following cases.

1. VMU is connected to new-generation game machine, and power to new-generation game machine is turned ON
2. Reset switch on VMU is pressed
3. Battery is inserted in VMU

Initialization comprises the following steps.

- 1) Clear main RAM
  - Write '00h' to entire main RAM area (bank 0, bank 1).

\* Initialization does not change XRAM values.

All registers are initialized by a hardware reset first, and then again by software. For information on the register values after a hardware reset, please refer to the VMU user's manual.

- 2) Set system clock and cycle time
  - Switch system clock to sub-clock (crystal quartz oscillator).
  - Set cycle time to 1/6 system clock.  
(The cycle time is used as reference for command execution. For details, please refer to the VMU user's manual.)
- 3) Set base timer
  - Select 14-bit base timer mode.
  - Switch base timer clock to sub-clock (crystal quartz oscillator).
  - Enable base timer 0 interrupt and start counting.

For details regarding base timer 0 operation, please refer to the VMU user's manual.

The base timer 0 is used by the clock function. For details regarding the clock function, please refer to Section , "Clock Function".

4) Set master interrupt

- Enable master interrupt.

(The master interrupt flag controls enabling/disabling of all interrupts with "High level" and "Low level" priority.)

5) Set LCD driver

- Activate LCD controller.
- Set LCD clock to 1/2 of LCD driver input clock.
- Set LCD start address to '000h' of XRAM.
- Set character register.
- Set time allocation register.
- Set LCD to ON.

6) Set port 1

- Set port 1 to all-bit input.
- Set bit 7 of port 1 to audio output pin.

\* After initialization, bit 7 of port 1 is set to input mode. Therefore a user program will need to again select the output mode.

- Set bit 5 – bit 0 of port 1 (serial interface for VMU) to synchronous operation. For details regarding the synchronous serial interface, please refer to the VMU user's manual.

7) Set port 3

- Pull up all bits of port 3.
- Set port 3 to all-bit input.
- Enable interrupt triggering and HOLD mode cancel by port 3.
- Enable interrupt trigger request by port 3.

8) Initialize Maple Bus interface circuit

- Initialize Maple Bus interface circuit.

9) Set work RAM

- Enable use of work RAM.

# Subroutine Description

This section describes the subroutines available in the System BIOS.

## Flash Memory Access Functions

The following subroutines are available for flash memory access.

- 1) Flash Memory Page Data Readout  
Read 128 bytes of data from the flash memory space.
- 2) Write to Flash Memory  
Write 128 bytes of data to the flash memory space.
- 3) Flash Memory Verify  
Verify data written to the flash memory.

\* When accessing the flash memory, the main clock in use must be switched to 600 kHz. For details, please refer to the next section.

### Precautions for Using Flash Memory Access Subroutines

When accessing the flash memory space, the following points must be observed.

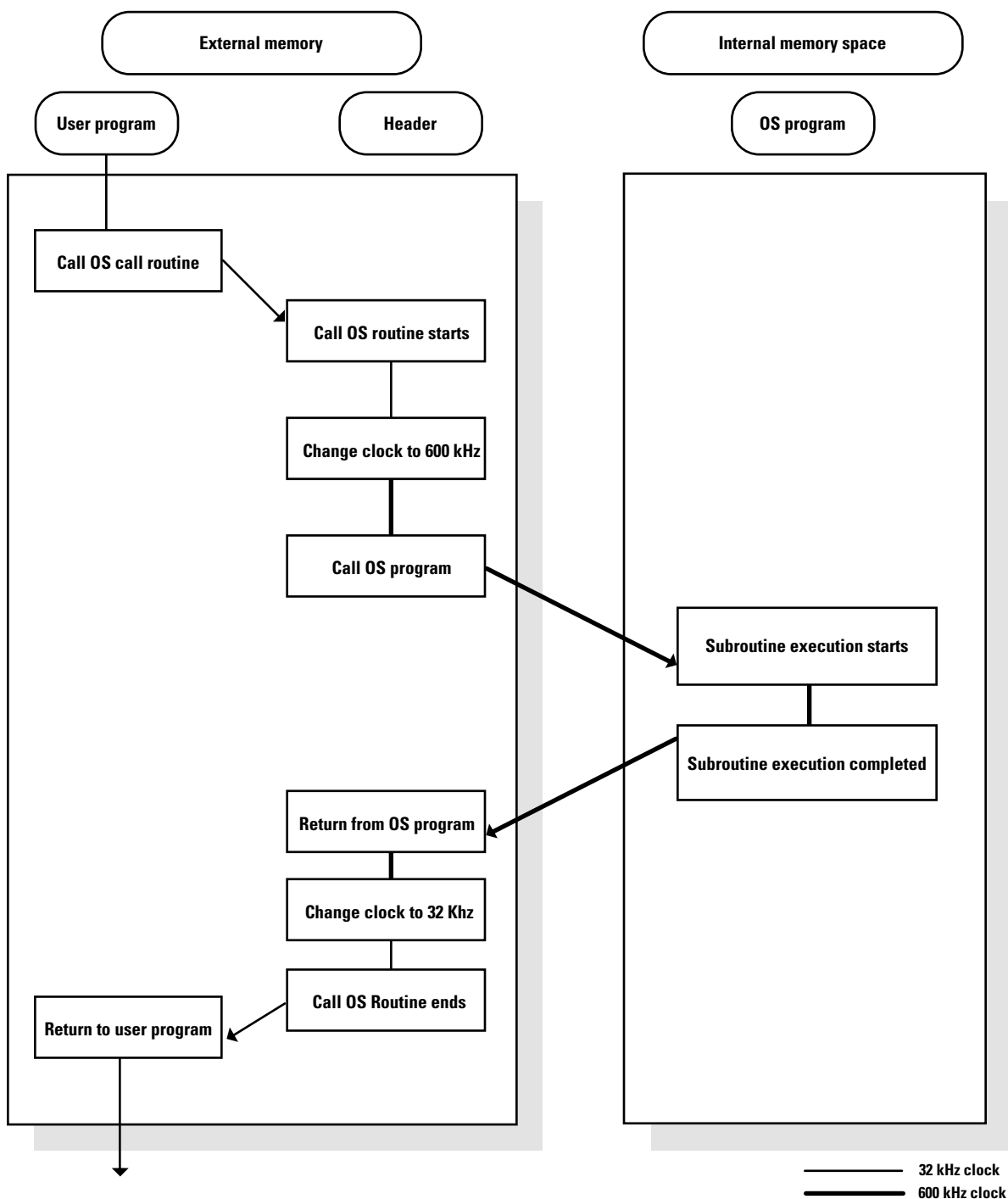
VMU uses three types of system clock as reference for command execution (see Figure 1.7, "System clock table").

When VMU is operating as a standalone unit, the quartz oscillator clock (32 kHz) will normally be used. However, for accessing the flash memory, the clock must be switched to the internal (RC) oscillator (600 kHz) before calling a flash memory access subroutine. After subroutine execution is completed, switch back to the previously used clock.

For information on the timing for clock switching, see Figure 1.7, "System clock table,".

System clock source	Oscillation frequency	Command cycle time
Ceramic (CF) oscillator	6 MHz	1.0 us
Internal (RC) oscillator	600 kHz	10.0 us
Quartz (X'TAL) oscillator	32 kHz	183.0 us

Figure 1.7 System clock table



**Figure 1.8** Flow diagram for clock switching during flash memory access

### Flash Memory Page Data Readout

Subroutine name:	fm_prd_ex (org 0120h)	
Arguments:	High-order start address for flash memory read:	fmadd_h (RAM bank-1 07Eh)
	Low-order start address for flash memory read:	fmadd_l (RAM bank-1 07Fh)
	Bank address for flash memory read:	fmbank (RAM bank-1 07Dh)
Return value:	Read data (128 bytes):	080h - 0FFh of RAM bank-1
Function:	Read one continuous page of data (128 bytes) from specified address	
Description:	By calling this subroutine, a program can read one page of data (128 bytes) from flash memory.	

Before using this subroutine, the following settings must be made.

- Select RAM bank to use
  - (1) Select RAM bank-1 (Set bit 1 of PSW to "1")  
For information on the PSW register, please refer to the VMU user's manual.
- Set start address for flash memory read
  - (2) High-order address (8 bit): set to fmadd\_h (07Eh of RAM bank-1)
  - (3) Low-order address (8 bit): set to fmadd\_l (07Fh of RAM bank-1)
- Select flash memory bank to read
  - (4) Select flash memory bank-0  
(Set 07Dh of RAM bank 1 to '00h')

\* If another value is set, normal operation is not assured.

The read data are written to 080h - 0FFh of RAM bank-1.

When making read settings, observe the following points.

- Data extending to 2 pages cannot be read. The read start address must therefore always be set to the beginning of each page.

The start address of each page can be calculated according to the following equation:

$$\text{start address value (2 byte)} = 080\text{h} \times \text{page number (0 - 511)}$$

(Because readout is performed in single-page units, bit 0 – bit 6 of the lower-level address must always be set to "0". If an address other than the start address of a page is set, normal operation is not assured.)

- The read-out data overwrite any previous content of the RAM.

\* Register values after subroutine completion

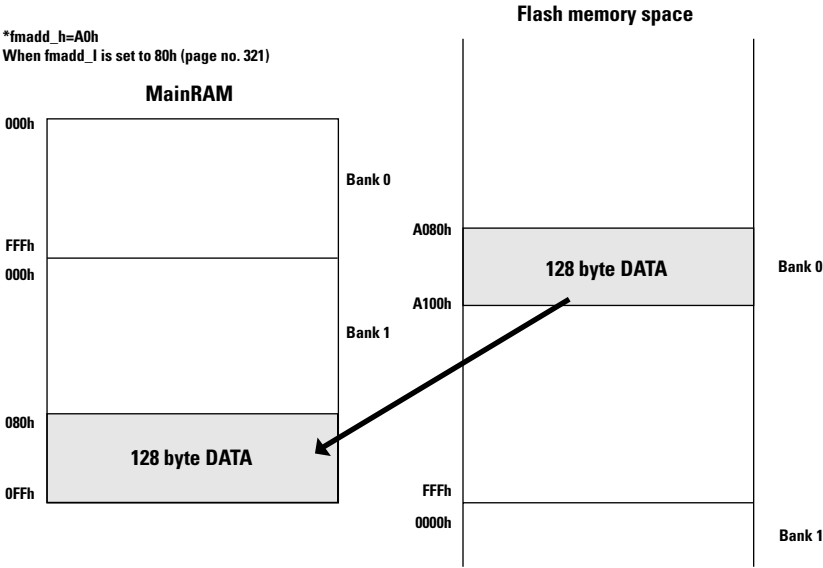
Note that the following (memory) registers will have different values before the subroutine is called and after the subroutine has completed:

- ACC (accumulator)
- TRL (table lookup register lower byte)
- TRH (table lookup register higher byte)
- r0 (RAM indirect address register)

\*About pages

Beginning at the top, the flash memory space is subdivided into 128-byte units called pages. Flash memory is managed in page units. Because 1 bank of the flash memory space is 64 kilobytes, it has 512 pages.

“fm\_prd\_ex” execution is shown in Figure 1.9, “Execution of fm\_prd\_ex,”.



**Figure 1.9** Execution of fm\_prd\_ex

### Writing to Flash Memory

Subroutine name: fm\_wrt\_ex (org 0100h)

Arguments: High-order start address for flash memory write: fmadd\_h (RAM bank-1 07Eh)  
Low-order start address for flash memory write: fmadd\_l (RAM bank-1 07Fh)  
Bank address for flash memory write: fmbank (RAM bank-1 07Dh)  
Flash memory write data (128 bytes): RAM bank-1 080h - 0FFh  
Data write end detection algorithm:  
Bit 0 of RAM bank-1 07Ch  
(toggle bit method (0)/ data polling method (1))

Return value: result of write: ACC (accumulator)  
(Normal termination: 00h. Abnormal termination: FFh)

Function: Write one continuous page of data (128 bytes) to the flash memory, starting at the specified address

Description: By calling this subroutine, a program can write a page of data (128 bytes) to a continuous area in the flash memory, starting at the specified address.

Before using this subroutine, the following settings must be made.

- Select RAM bank to use
  - (1) Select RAM bank 1 (Set bit 1 of PSW to "1")
- Prepare data to be written to flash memory
  - (2) Store data to be written to flash memory in RAM bank 1, 080h - 0FFh
- Select flash memory bank to read
  - (3) Select flash memory bank 0  
(Set 07Dh of RAM bank 1 to '00h')

\* If another value is set, normal operation is not assured.
- Set address for accessing flash memory
  - (4) High-order address (8 bit): set to 07Eh of RAM bank-1
  - (5) Low-order address (8 bit): set to 07Fh of RAM bank-1
- Specify data write end detection algorithm
  - (6) Set data write end detection algorithm in 07Ch of RAM bank-1, as follows.
    - (6-1) Use toggle bit method: set 07Ch to 00h
    - (6-2) Use data polling method: set 07Ch to 01h

\* If another value is set, normal operation is not assured.

When making write settings, observe the following points.

- fm\_wrt\_ex is a subroutine specifically for user programs. This subroutine can write only to the area where the user program is located. For this reason, be sure to secure an area within the user program before performing the data write.
- Data extending to 2 pages cannot be written. The write start address must therefore always be set to the beginning of each page.

The start address of each page can be calculated according to the following equation:  
start address value (2 byte) = 080h x page number (0 - 511)



(Because writing is performed in single-page units, bit 0 - 6 of the lower-level address must always be set to "0". If an address other than the start address of a page is set, normal operation is not assured.)

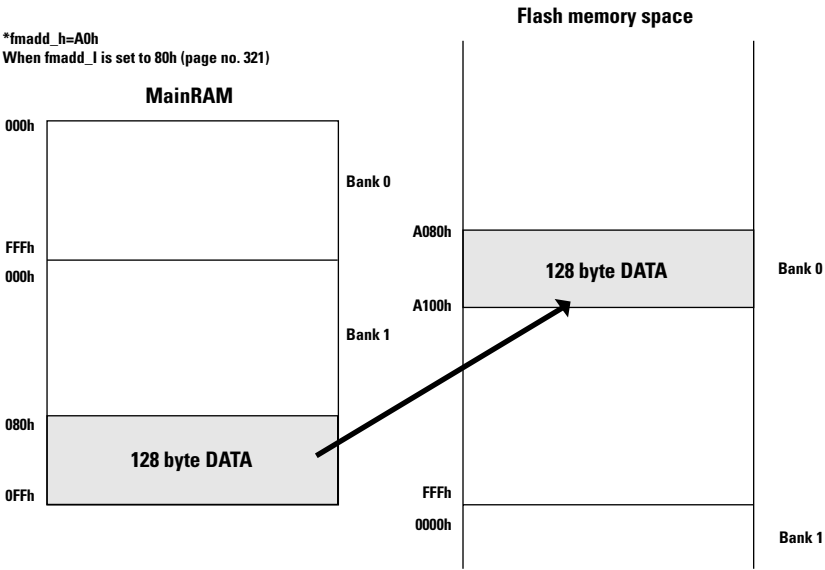
For information on pages, please refer to Section , "Flash Memory Page Data Readout".

\* Register values after subroutine completion

Note that the following (memory) registers will have different values before the subroutine is called and after the subroutine has completed:

- ACC (accumulator)
- B (B register)
- C (C register)
- TRL (table lookup register lower byte)
- TRH (table lookup register higher byte)
- r0 (RAM indirect access register)

fm\_wrt\_ex execution is shown in Figure 1.10, "Execution of fm\_wrt\_ex,".



**Figure 1.10** Execution of `fm_wrt_ex`

## Flash Memory Verify

Subroutine name: `fm_vrf_ex` (org 0110h)

Arguments: High-order address flash memory address for verify start: `fmadd_h` (RAM bank 1 07Eh)  
 Low-order address flash memory address for verify start: `fmadd_l` (RAM bank 1 07Fh)  
 Flash memory bank address for verify operation: `fmbank` (RAM bank 1 07Dh)  
 Data (128 bytes) for verify operation: RAM bank 1 080h - 0FFh

Return value: Verify result: accumulator (ACC) (normal end: 00h?error end: other than 00h)

Function: Serves to verify whether data were written correctly to flash memory. For use after writing data to flash memory with `fm_wrt_ex` (see section 7.1.4).

Description: This subroutine compares the 128 byte data specified when calling `fm_wrt_ex` with the data actually written to flash memory. Therefore the subroutine can only be used immediately after the `fm_wrt_ex` subroutine was called.

When calling this subroutine, the same arguments as used for the preceding `fm_wrt_ex` must be specified. If different arguments are specified, data verify will not be carried out properly.

After calling this subroutine, if all 128 bytes of data were found to match, 00h will be set in ACC, and the routine returns. If a data mismatch was detected, a value other than 00h will be set in ACC, and the routine returns.

\* Register values after subroutine completion

Note that the following (memory) registers will have different values before the subroutine is called and after the subroutine has completed.

- TRL (table lookup register lower byte)
- TRH (table lookup register higher byte)
- r0 (RAM indirect access register)

`fm_vrf_ex` execution is shown in Figure 1.11, "Execution of `fm_vrf_ex`".

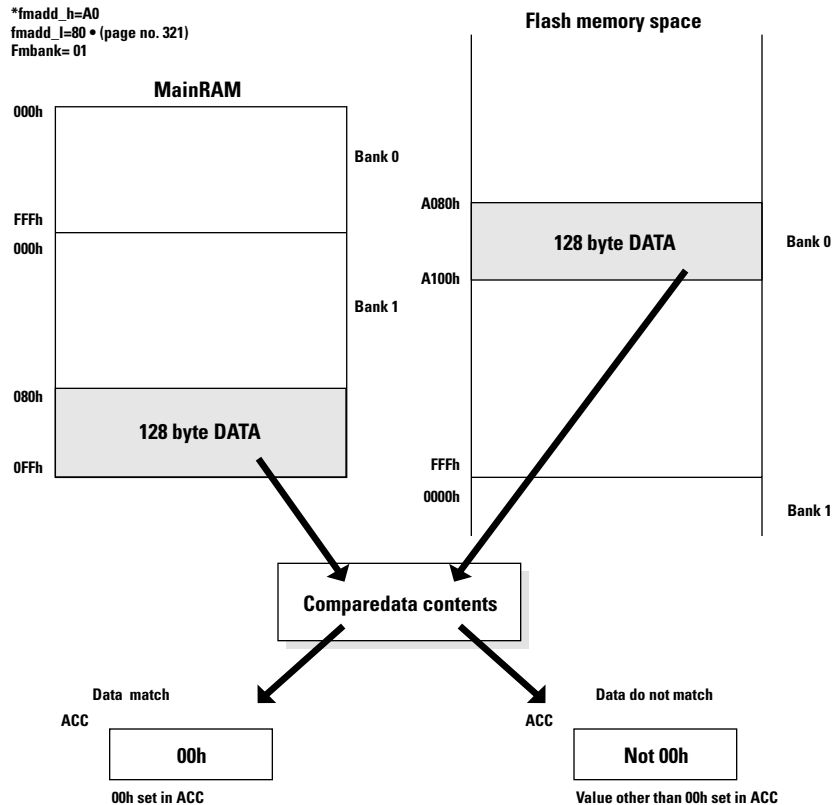


Figure 1.11 Execution of `fm_vrf_ex`

## Clock Function

The clock functions implemented in VMU are as follows.

Time data automatic update

### Clock Countup Timer

Subroutine name: timer\_ex

Arguments: None

Return value:

Year:	year_h	(RAM bank 0 017h, 18h)
Month:	mon_h	(RAM bank 0 019h)
Day:	day_h	(RAM bank 0 01Ah)
Hour:	hour_h	(RAM bank 0 01Bh)
Minute:	min_h	(RAM bank 0 01Ch)
Second:	sec_h	(RAM bank 0 01Dh)

\* The year data are configured as 2 bytes, with the higher-level in byte in 17h and the lower level byte in 18h. Because "year\_h" is assigned to RAM bank, 017h, address 018h can be accessed by specifying "year\_h+1".

Function: When the subroutine is called, it obtains the time data and places them in the specified location in RAM bank 0. (For information on the specified location, please refer to Appendix 1.)

Description: This subroutine is a time counter using the base timer interrupt. To enable use of the subroutine, the following settings for the base timer interrupt must be made.

- Base timer interrupt settings

This subroutine uses only the base timer 0 interrupt. The base timer interrupt is to be set as shown below.

- |  |                    |
|--|--------------------|
| (1) Base timer count stop              | (BTCR 6 bit = '0') |
| (2) 14 bit base timer mode selected    | (BTCR 7 bit = '0') |
| (3) Sub clock used as base timer clock | (ISL 4 bit = '0')  |
| (4) Base timer interrupt 0 enabled     | (BTCR 0 bit = '1') |
| (5) Base timer count start             | (BTCR 6 bit = '1') |

Because the base timer 0 interrupt is used by the timer\_ex subroutine, user programs may not access this interrupt. Otherwise, normal operation is not assured.

This subroutine should be called after jumping to the interrupt vector of the base timer interrupt 0 source. Also, be sure to clear the base timer 0 interrupt source (BTCR 1 bit = '0').

(If this is not performed, the clock function will not operate properly.)

All time data obtained by this subroutine are in hex format. Conversion into decimal format must be performed by the user program.

# **Automatic low battery detection function**

Visual Memory comes with the ability to automatically detect low battery.

The following explains how this function works.

## **Automatic low battery detection flag**

Visual Memory can automatically check the battery's power consumption and when necessary display a low battery warning message on the screen. Gamedevelopers can use the automatic low battery detection flag to enable or disable this function.

The following describes how to use this function.

Register to use:	06Eh (Bank-0)
Register values:	00h (enable the automatic low battery detection function) FFh (disable the automatic low battery detection function) (If any value other than the above ones is used, then normal operation cannot be guaranteed.)
How it work:	The automatic low battery detection function constantly monitors the battery's voltage and if the voltage falls below a certain level it will stop the current program, wait for 3 seconds, then display the battery warning message on the screen.
Explanation:	The automatic low battery detection function consists of tasks from detecting low voltage to displaying the low battery warning message. When the automatic low battery detection flag is set to 00h, the automatic low battery detection function is enabled and when the battery is low it will display the low battery warning message, regardless of the current task of Visual Memory. If the flag is set to FFh, then the automatic low battery detection function is disabled.

When the user program is performing the following tasks, be sure to turn off the automatic low battery detection function:

1. Communicating with another Visual Memory via the serial interface
2. Writing to the flash memory space





***Visual Memory Unit (VMU)***  
***Sound Development***  
***Specifications***





# ***Table of Contents***

<b>VMU Sound Development Specifications .....</b>	<b>VMA-1</b>
VMU Sound Output Hardware Outline .....	VMA-1
Sound Output Principle .....	VMA-2
Timer 1 Outline .....	VMA-2
8-Bit Counter Mode .....	VMA-5
Table of Available Output Frequencies .....	VMA-8
Sample Program .....	VMA-13



# ***VMU Sound Development Specifications***

---

## **VMU Sound Output Hardware Outline**

VMU can use an internal timer (timer 1) to produce sound output.

The following two output methods are possible.

- 8-bit pulse generator output
- Variable bit length pulse generator output (9 - 16bits)

Both methods use the timer 1 circuit. Normally, the 8-bit pulse generator output method is used.

## Sound Output Principle

This section describes the VMU sound output method.

VMU sound output uses timer 1.

### Timer 1 Outline

This section describes timer 1 that is used for VMU sound output.

Timer 1 incorporated in the VMU is a 16-bit timer with the following four functions.

Mode 0: 8-bit reload timer x 2 channels

Mode 1: 8-bit reload timer + 8-bit pulse generator

Mode 2: 16-bit reload timer

Mode 3: Variable bit length pulse generator (9 - 16bits)

Among these modes, VMU uses mode 1 for sound output.

For information on using the other modes, please refer to the VMU Hardware manual.

#### Timer 1 Block Configuration

This section describes the block configuration of timer 1.

A configuration diagram of timer 1 is shown in Figure 1.1, "VMU Timer 1 Block Diagram,".

- Timer 1 lower level (T1L) ..... 1

This is an 8-bit reload timer using the cycle clock or cycle clock 1/2 signal as clock signal.

At the overflow of T1L, the T1LR data are reloaded. When T1LRUN (T1CNT, bit6) is set to "0", the T1LR data are transferred to T1L.

- Timer 1 lower level comparator (T1LC) ..... 2

This circuit consists of the 8-bit timer 1 lower level comparison data register (T1LC) and an 8-bit data comparator circuit. The circuit compares the T1L and T1LC data.

- Timer 1 higher level (T1H) ..... 3

This is an 8-bit reload timer using the cycle clock or the T1L overflow as clock signal.

At the overflow of T1H, the T1HR data are reloaded. Reload also occurs when T1HRUN (T1CNT, bit7) is reset.

- Timer 1 higher level comparator (T1HC) ..... 4

This circuit consists of the 8-bit timer 1 higher level comparison data register (T1HC) and an 8-bit data comparator circuit. The circuit compares the T1H and T1HC data.

- Timer 1 control register (T1CNT) ..... 5

Serves for T1 mode setting and interrupt control.

# Visual Memory Unit (VMU) Sound Development Specifications

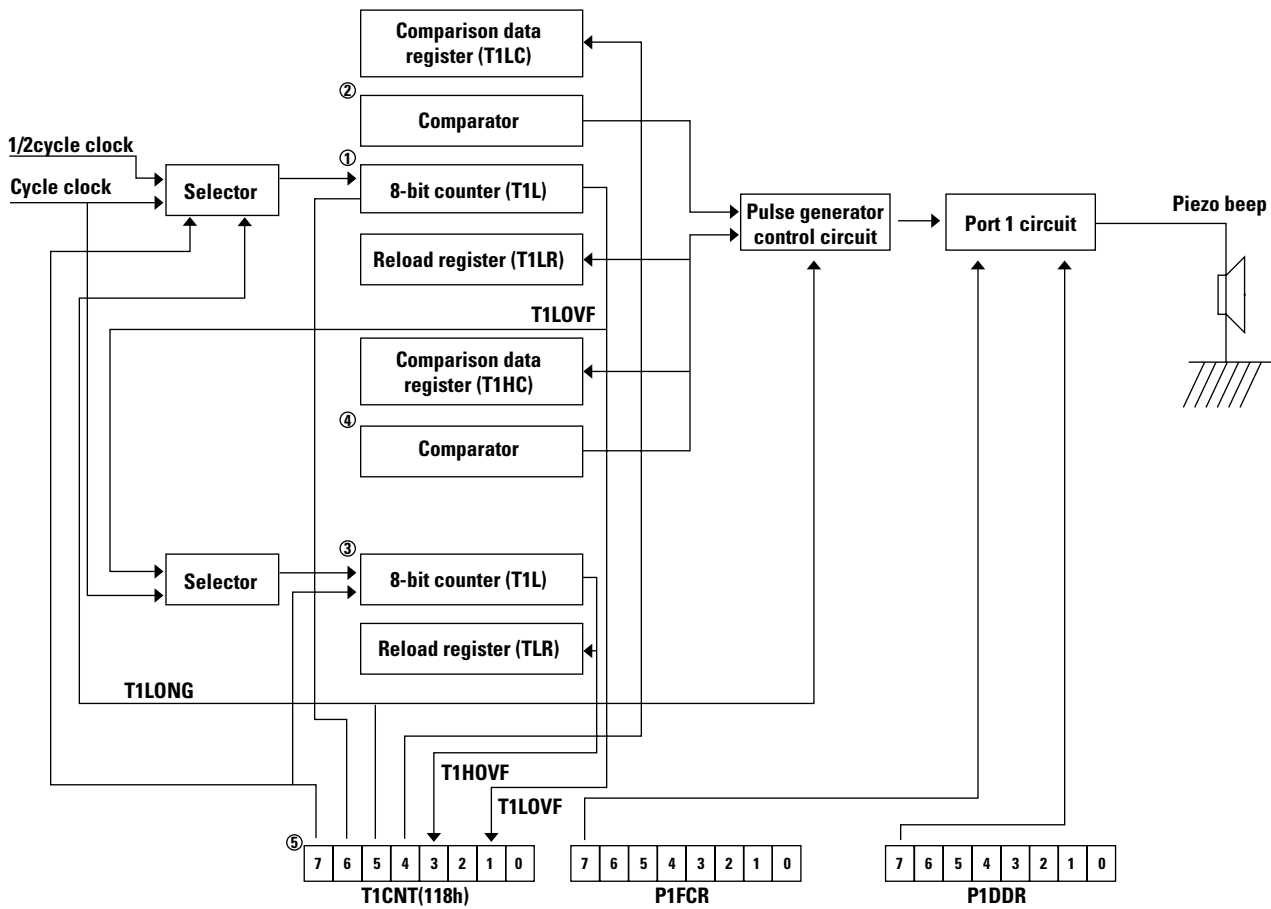


Figure 1.1 VMU Timer 1 Block Diagram

## Related Registers

The following registers are required for controlling timer 1.

- T1L (11Bh)                      Timer 1 lower level counter register
- T1LR (11Bh)                    Timer 1 lower level reload register
- T1LC (11Ah)                    Timer 1 lower level comparison data register
- T1CNT (118h)                  Timer 1 control register
- P1 (114h)                        Port 1 latch register
- P1DDR (145h)                  Port 1 data direction register
- P1FCR (146h)                  Port 1 control register
- OCR (10Eh)                     Resonance control register

For details on the above timer, please refer to the timer section of the VMU Hardware manual.

## Mode Setting

This section describes how to set timer 1 to the mode for VMU sound output (mode 1).

The following four registers are required for setting the mode.

T1CNT	(bit5: T1LONG)
P1	(bit7: P17)
P1DDR	(bit7: P17DDR)
P1FCR	(bit7: P17DDR)

The register values for the modes are listed in the table below, along with the cycle clock used for each mode.

Mode	Clock cycle	T1LONG	P17FCR	P17DDR	P17
1	Tcyc	0	1	1	0

**Table 1.1 Time 1 Mode Setting**

Tcyc in the table is the cycle clock.

To use the sound output capability of VMU, be sure to set the system clock to the sub-clock (32 KHz).

At other system clock settings, correct sound output may not be obtained.

The cycle clock is defined as follows.

System clock 32 KHz (Tcyc = 183.0 us)

For information on setting the system clock, please refer to the VMU Hardware manual.

\* Problems when using other system clock settings

Besides the 32 KHz clock, the VMU can use a 600 KHz and 6 MHz system clock, but when the latter two are selected, the following problems occur.

- 600KHz When the 600 KHz clock is selected, the output frequency tolerance will be -50%, +100%, which will cause a wide fluctuation in the actual output sound.
- 6MHz When the 6 MHz clock is selected, power consumption will increase considerably, resulting in a shorter battery life.

## 8-Bit Counter Mode

This section describes VMU sound output when using 8-bit counter mode. For information on basic operation, please refer to the VMU Hardware manual.

### Output Waveform and Parameter Settings

This section describes the signal waveform that can be output in 8-bit counter mode, and lists the parameters that determine the waveform.

The output waveform is shown in Figure 1.2, "Output waveform,".

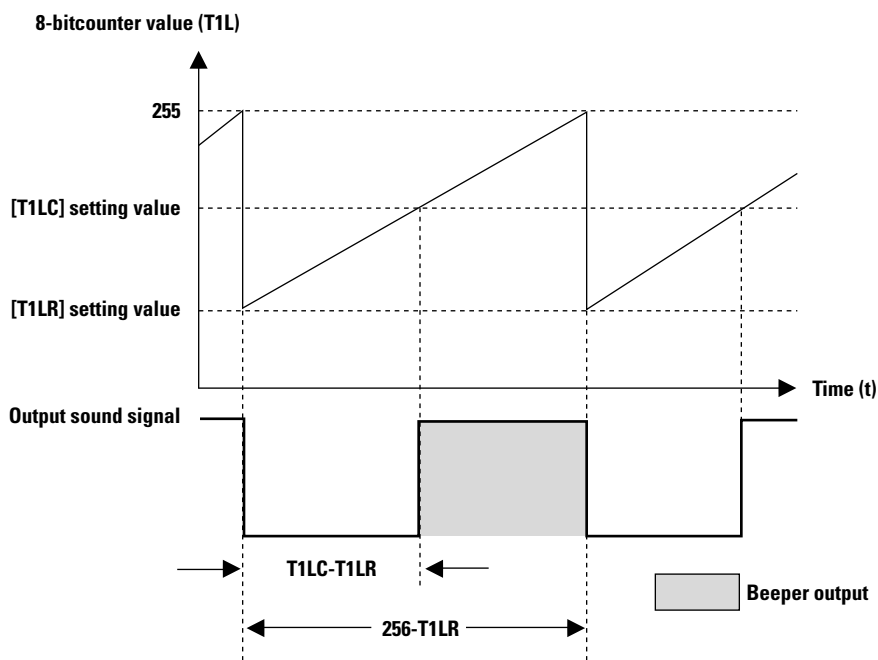


Figure 1.2 Output waveform

**8-Bit Counter Mode Setting**

This section describes the sound signal output procedure in 8-bit counter mode.

To output a sound signal in 8-bit counter mode, make the settings as described below.

1. Set the parameters (T1LR, T1LC) according to the desired output waveform.

Use equations (1) and (2) given below to define the waveform.

$$\begin{aligned} &\text{Sound output signal L level pulse width (decimal)} \\ &= (\text{T1LC setting value} - \text{T1LR setting value}) \times \text{Tcyc} \dots \text{Equation (1)} \end{aligned}$$

$$\begin{aligned} &\text{Sound output signal cycle (decimal)} \\ &= (256 - \text{T1LR setting value}) \times \text{Tcyc} \dots \text{Equation (2)} \end{aligned}$$

Tcyc: cycle clock

2. Select the mode for timer 1.

The following four registers are required for setting the mode.

- T1CNT            (bit5: T1LONG)
- P1                (bit7: P17)
- P1DDR           (bit7: P17DDR)
- P1FCR           (bit7: P17FCR)

The register values for the modes are listed in the table below, along with the cycle clock used for each mode.

**Table 1.2 Time 1 Mode Setting**

Mode	T1LONG	P17FCR	P17DDR	P17
1	0	1	1	0



3. Start the count for timer 1 (lower 8bits)

To start/stop the timer, make the following settings.

Waveform parameter update

Set T1CNT bit4 (ELDT1C) to "1". Note that the waveform parameters set in step 1 do not become effective until this setting is made.

If the parameters were changed while T1CNT bit4 was "1", the parameter setting value becomes effective immediately after the change.

Timer 1 count start

Set T1CNT bit6 (T1LRUN) to "1".

To stop audio output in the 8-bit counter made, make the Following setting.

4. Set the timer1(T1L) count stop flag (T1CNT bit6)to "0".

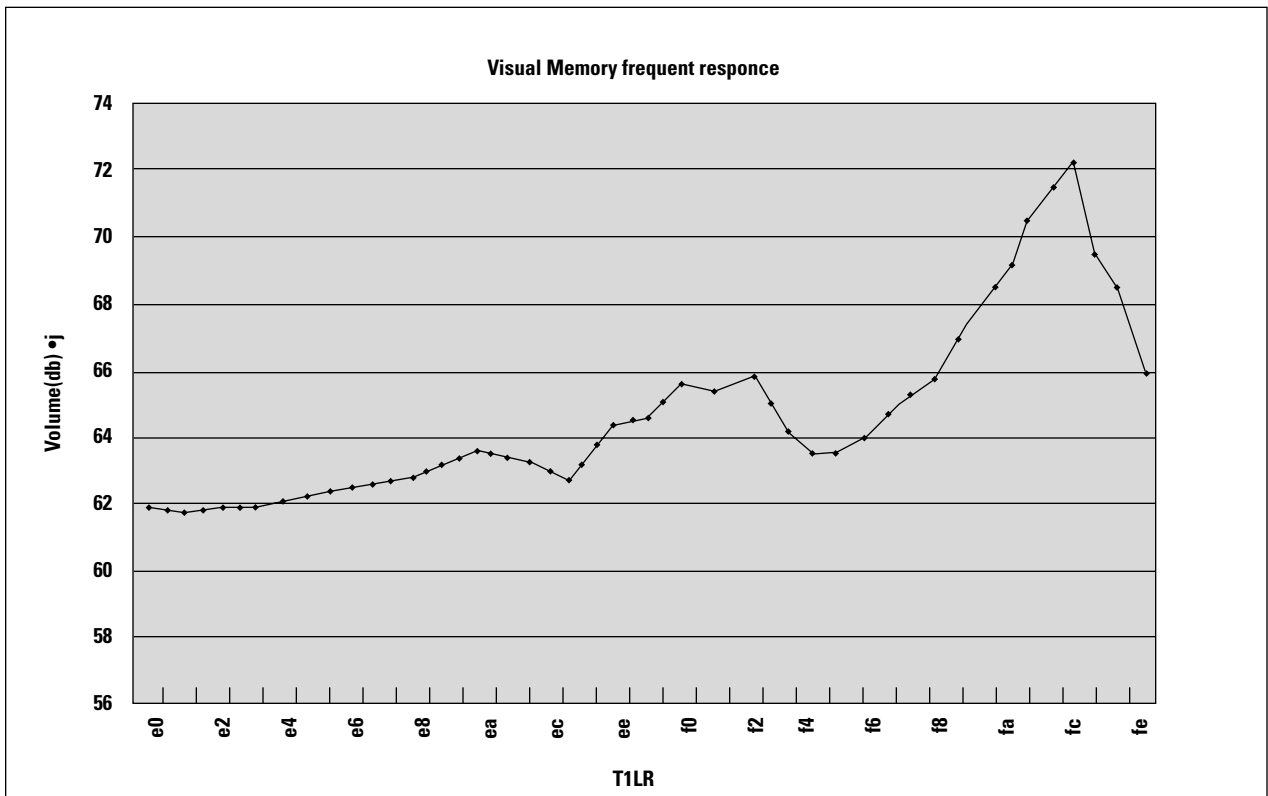
While timer 1 (lower 8bits) is operating, the waveform parameters can be changed. To output sound of a different frequency without interruption, change the waveform output parameters without stopping timer 1. (Leave T1CNT bit4 [ELDT 1C]) set to "1".)

**Frequency Response Characteristics**

The frequency response of the beeper in the VMU is shown below.

The T1LR value indicates the frequency range that can be output by the VMU.

For details, please refer to the explanation of the relationship between T1LR value and output frequency in section on "Table of Available Output Frequencies" on page 8.



## Visual Memory Unit (VMU) Sound Development Specifications

### Table of Available Output Frequencies

The output frequencies (theoretical values) available with a system clock of 32 KHz are listed below.

Due to limitations of the beeper, not all frequencies can actually be output. You should use the recommended frequencies indicated in the table.

The L level pulse width of the output signal is set to 1/2 of the output signal cycle (duty factor = 50%).

**Table 1.3** *Waveform Parameters and Output Frequencies*

T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)
00	80	21.346	40	94	28.461	80	A8	42.691	C0	E0	85.383
01	80	21.429	41	A0	28.610	81	C0	43.027	C1	E0	86.738
02	81	21.514	42	A1	28.760	82	C1	43.369	C2	E1	88.137
03	81	21.599	43	A1	28.913	83	C1	43.716	C3	E1	89.582
04	82	21.684	44	A2	29.066	84	C2	44.068	C4	E2	91.075
05	82	21.771	45	A2	29.222	85	C2	44.427	C5	E2	92.618
06	83	21.858	46	A3	29.379	86	C3	44.791	C6	E3	94.215
07	83	21.946	47	A3	29.538	87	C3	45.161	C7	E3	95.868
08	84	22.034	48	A4	29.698	88	C4	45.537	C8	E4	97.580
09	84	22.123	49	A4	29.861	89	C4	45.920	C9	E4	99.354
0A	85	22.213	4A	A5	30.025	8A	C5	46.309	CA	E5	101.194
0B	85	22.304	4B	A5	30.191	8B	C5	46.705	CB	E5	103.103
0C	86	22.395	4C	A6	30.358	8C	C6	47.108	CC	E6	105.086
0D	86	22.488	4D	A6	30.528	8D	C6	47.517	CD	E6	107.147
0E	87	22.580	4E	A7	30.699	8E	C7	47.934	CE	E7	109.290
0F	87	22.674	4F	A7	30.873	8F	C7	48.358	CF	E7	111.520
10	88	22.769	50	A8	31.048	90	C8	48.790	D0	E8	113.843
11	88	22.864	51	A8	31.226	91	C8	49.230	D1	E8	116.266
12	89	22.960	52	A9	31.405	92	C9	49.677	D2	E9	118.793
13	89	23.057	53	A9	31.587	93	C9	50.133	D3	E9	121.433
14	8A	23.155	54	AA	31.770	94	CA	50.597	D4	EA	124.193
15	8A	23.253	55	AA	31.956	95	CA	51.070	D5	EA	127.081
16	8B	23.352	56	AB	32.144	96	CB	51.552	D6	EB	130.107

## VMU Sound Development Specifications

T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)
17	8B	23.453	57	AB	32.334	97	CB	52.043	D7	EB	133.280
18	8C	23.554	58	AC	32.527	98	CC	52.543	D8	EC	136.612
19	8C	23.656	59	AC	32.721	99	CC	53.053	D9	EC	140.115
1A	8D	23.759	5A	AD	32.919	9A	CD	53.573	DA	ED	143.802
1B	8D	23.862	5B	AD	33.118	9B	CD	54.104	DB	ED	147.689
1C	8E	23.967	5C	AE	33.320	9C	CE	54.645	DC	EE	151.791
1D	8E	24.073	5D	AE	33.524	9D	CE	55.197	DD	EE	156.128
1E	8F	24.179	5E	AF	33.731	9E	CF	55.760	DE	EF	160.720
1F	8F	24.287	5F	AF	33.941	9F	CF	56.335	DF	EF	165.590
20	90	24.395	60	B0	34.153	A0	D0	56.922	E0	F0	170.765
21	90	24.504	61	B0	34.368	A1	D0	57.521	E1	F0	176.274
22	91	24.615	62	B1	34.585	A2	D1	58.133	E2	F1	182.149
23	91	24.726	63	B1	34.806	A3	D1	58.758	E3	F1	188.430
24	92	24.839	64	B2	35.029	A4	D2	59.397	E4	F2	195.160
25	92	24.952	65	B2	35.255	A5	D2	60.049	E5	F2	202.388
26	93	25.066	66	B3	35.484	A6	D3	60.716	E6	F3	210.172
27	93	25.182	67	B3	35.716	A7	D3	61.399	E7	F3	218.579
28	94	25.299	68	B4	35.951	A8	D4	62.096	E8	F4	227.687
29	94	25.416	69	B4	36.189	A9	D4	62.810	E9	F4	237.586
2A	95	25.535	6A	B5	36.430	AA	D5	63.540	EA	F5	248.385
2B	95	25.655	6B	B5	36.674	AB	D5	64.288	EB	F5	260.213
2C	96	25.776	6C	B6	36.922	AC	D6	65.053	EC	F6	273.224
2D	96	25.898	6D	B6	37.173	AD	D6	65.837	ED	F6	287.604
2E	97	26.021	6E	B7	37.428	AE	D7	66.640	EE	F7	303.582
2F	97	26.146	6F	B7	37.686	AF	D7	67.463	EF	F7	321.440
30	98	26.272	70	B8	37.948	B0	D8	68.306	F0	F8	341.530
31	98	26.398	71	B8	38.213	B1	D8	69.171	F1	F8	364.299
32	99	26.527	72	B9	38.482	B2	D9	70.057	F2	F9	390.320
33	99	26.656	73	B9	38.755	B3	D9	70.967	F3	F9	420.345

## Visual Memory Unit (VMU) Sound Development Specifications

T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)
34	9A	26.787	74	BA	39.032	B4	DA	71.901	F4	FA	455.373
35	9A	26.919	75	BA	39.313	B5	DA	72.860	F5	FA	496.771
36	9B	27.052	76	BB	39.598	B6	DB	73.844	F6	FB	546.448
37	9B	27.186	77	BB	39.887	B7	DB	74.856	F7	FB	607.165
38	9C	27.322	78	BC	40.180	B8	DC	75.896	F8	FC	683.060
39	9C	27.460	79	BC	40.478	B9	DC	76.965	F9	FC	780.640
3A	9D	27.598	7A	BD	40.780	BA	DD	78.064	FA	FD	910.747
3B	9D	27.738	7B	BD	41.086	BB	DD	79.195	FB	FD	1092.896
3C	9E	27.880	7C	BE	41.398	BC	DE	80.360	FC	FE	1366.120
3D	9E	28.023	7D	BE	41.714	BD	DE	81.559	FD	FE	1821.494
3E	9F	28.167	7E	BF	42.034	BE	DF	82.795	FE	FF	2732.240
3F	9F	28.313	7F	BF	42.360	BF	DF	84.069	FF	FF	5464.481
00	80	21.346	40	94	28.461	80	A8	42.691	C0	E0	85.383
01	80	21.429	41	A0	28.610	81	C0	43.027	C1	E0	86.738
02	81	21.514	42	A1	28.760	82	C1	43.369	C2	E1	88.137
03	81	21.599	43	A1	28.913	83	C1	43.716	C3	E1	89.582
04	82	21.684	44	A2	29.066	84	C2	44.068	C4	E2	91.075
05	82	21.771	45	A2	29.222	85	C2	44.427	C5	E2	92.618
06	83	21.858	46	A3	29.379	86	C3	44.791	C6	E3	94.215
07	83	21.946	47	A3	29.538	87	C3	45.161	C7	E3	95.868
08	84	22.034	48	A4	29.698	88	C4	45.537	C8	E4	97.580
09	84	22.123	49	A4	29.861	89	C4	45.920	C9	E4	99.354
0A	85	22.213	4A	A5	30.025	8A	C5	46.309	CA	E5	101.194
0B	85	22.304	4B	A5	30.191	8B	C5	46.705	CB	E5	103.103
0C	86	22.395	4C	A6	30.358	8C	C6	47.108	CC	E6	105.086
0D	86	22.488	4D	A6	30.528	8D	C6	47.517	CD	E6	107.147
0E	87	22.580	4E	A7	30.699	8E	C7	47.934	CE	E7	109.290
0F	87	22.674	4F	A7	30.873	8F	C7	48.358	CF	E7	111.520
10	88	22.769	50	A8	31.048	90	C8	48.790	D0	E8	113.843

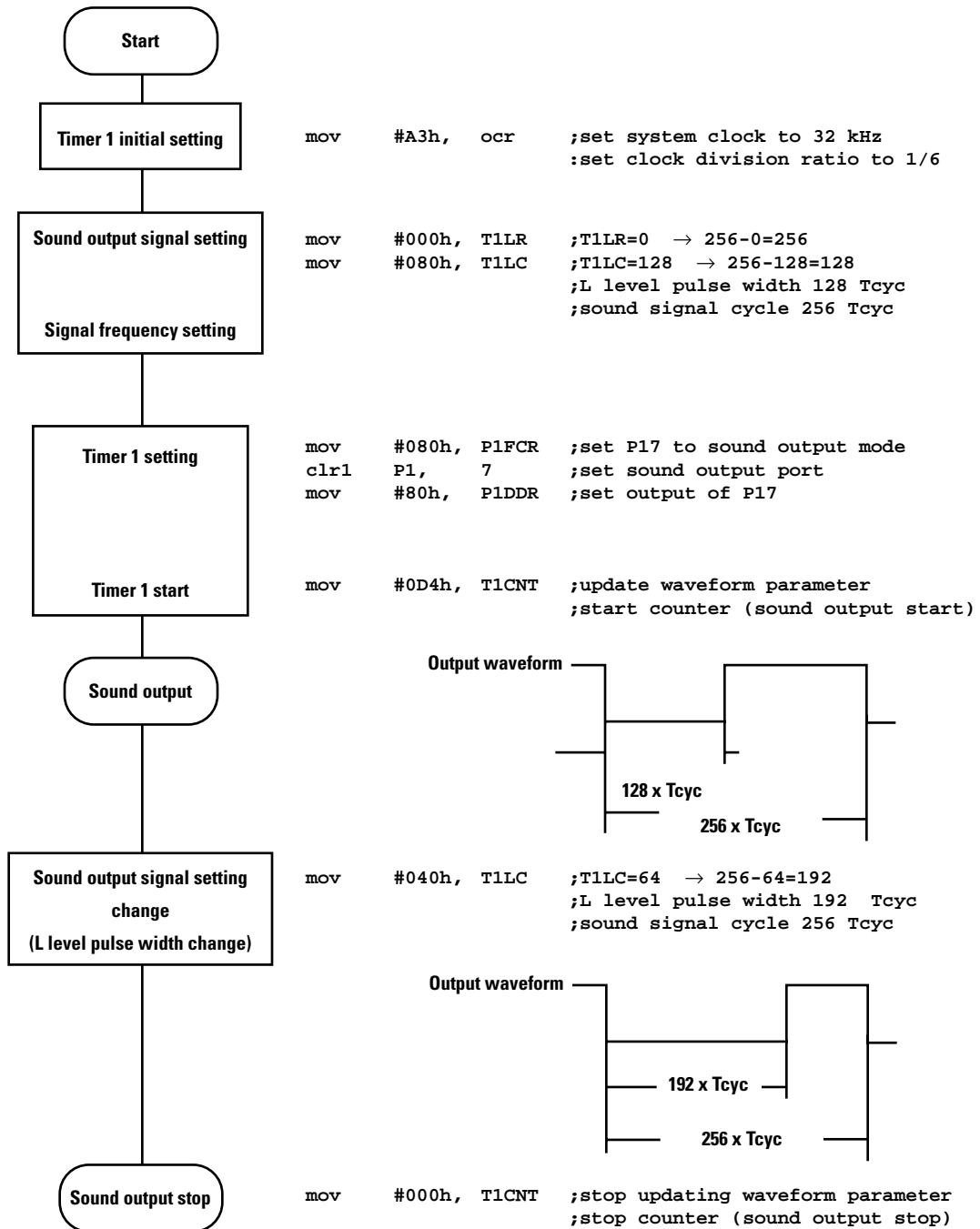
## VMU Sound Development Specifications

T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)
11	88	22.864	51	A8	31.226	91	C8	49.230	D1	E8	116.266
12	89	22.960	52	A9	31.405	92	C9	49.677	D2	E9	118.793
13	89	23.057	53	A9	31.587	93	C9	50.133	D3	E9	121.433
14	8A	23.155	54	AA	31.770	94	CA	50.597	D4	EA	124.193
15	8A	23.253	55	AA	31.956	95	CA	51.070	D5	EA	127.081
16	8B	23.352	56	AB	32.144	96	CB	51.552	D6	EB	130.107
17	8B	23.453	57	AB	32.334	97	CB	52.043	D7	EB	133.280
18	8C	23.554	58	AC	32.527	98	CC	52.543	D8	EC	136.612
19	8C	23.656	59	AC	32.721	99	CC	53.053	D9	EC	140.115
1A	8D	23.759	5A	AD	32.919	9A	CD	53.573	DA	ED	143.802
1B	8D	23.862	5B	AD	33.118	9B	CD	54.104	DB	ED	147.689
1C	8E	23.967	5C	AE	33.320	9C	CE	54.645	DC	EE	151.791
1D	8E	24.073	5D	AE	33.524	9D	CE	55.197	DD	EE	156.128
1E	8F	24.179	5E	AF	33.731	9E	CF	55.760	DE	EF	160.720
1F	8F	24.287	5F	AF	33.941	9F	CF	56.335	DF	EF	165.590
20	90	24.395	60	B0	34.153	A0	D0	56.922	E0	F0	170.765
21	90	24.504	61	B0	34.368	A1	D0	57.521	E1	F0	176.274
22	91	24.615	62	B1	34.585	A2	D1	58.133	E2	F1	182.149
23	91	24.726	63	B1	34.806	A3	D1	58.758	E3	F1	188.430
24	92	24.839	64	B2	35.029	A4	D2	59.397	E4	F2	195.160
25	92	24.952	65	B2	35.255	A5	D2	60.049	E5	F2	202.388
26	93	25.066	66	B3	35.484	A6	D3	60.716	E6	F3	210.172
27	93	25.182	67	B3	35.716	A7	D3	61.399	E7	F3	218.579
28	94	25.299	68	B4	35.951	A8	D4	62.096	E8	F4	227.687
29	94	25.416	69	B4	36.189	A9	D4	62.810	E9	F4	237.586
2A	95	25.535	6A	B5	36.430	AA	D5	63.540	EA	F5	248.385
2B	95	25.655	6B	B5	36.674	AB	D5	64.288	EB	F5	260.213
2C	96	25.776	6C	B6	36.922	AC	D6	65.053	EC	F6	273.224
2D	96	25.898	6D	B6	37.173	AD	D6	65.837	ED	F6	287.604

## Visual Memory Unit (VMU) Sound Development Specifications

T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)	T1LR(hex)	T1LC(hex)	Frequency (Hz)
2E	97	26.021	6E	B7	37.428	AE	D7	66.640	EE	F7	303.582
2F	97	26.146	6F	B7	37.686	AF	D7	67.463	EF	F7	321.440
30	98	26.272	70	B8	37.948	B0	D8	68.306	F0	F8	341.530
31	98	26.398	71	B8	38.213	B1	D8	69.171	F1	F8	364.299
32	99	26.527	72	B9	38.482	B2	D9	70.057	F2	F9	390.320
33	99	26.656	73	B9	38.755	B3	D9	70.967	F3	F9	420.345
34	9A	26.787	74	BA	39.032	B4	DA	71.901	F4	FA	455.373
35	9A	26.919	75	BA	39.313	B5	DA	72.860	F5	FA	496.771
36	9B	27.052	76	BB	39.598	B6	DB	73.844	F6	FB	546.448
37	9B	27.186	77	BB	39.887	B7	DB	74.856	F7	FB	607.165
38	9C	27.322	78	BC	40.180	B8	DC	75.896	F8	FC	683.060
39	9C	27.460	79	BC	40.478	B9	DC	76.965	F9	FC	780.640
3A	9D	27.598	7A	BD	40.780	BA	DD	78.064	FA	FD	910.747
3B	9D	27.738	7B	BD	41.086	BB	DD	79.195	FB	FD	1092.896
3C	9E	27.880	7C	BE	41.398	BC	DE	80.360	FC	FE	1366.120
3D	9E	28.023	7D	BE	41.714	BD	DE	81.559	FD	FE	1821.494
3E	9F	28.167	7E	BF	42.034	BE	DF	82.795	FE	FF	2732.240
3F	9F	28.313	7F	BF	42.360	BF	DF	84.069	FF	FF	5464.481

# Sample Program







***Visual Memory Unit (VMU)***  
***Simulator Manual***



# Table of Contents

<b>Overview .....</b>	<b>VMB-1</b>
Features .....	VMB-1
Visual Memory Simulator Operating Environment .....	VMB-2
Checking Operation on Actual Visual Memory Hardware .....	VMB-3
Notes Concerning Startup for the First Time .....	VMB-4
<b>Implemented Devices .....</b>	<b>VMB-5</b>
Virtual CPU .....	VMB-5
Memory .....	VMB-6
LCD Controller (LCDC) .....	VMB-6
Serial Interface (SIO) .....	VMB-7
Timer .....	VMB-7
Interrupt Controller .....	VMB-7
I/O Ports .....	VMB-7
External Input Devices .....	VMB-8
<b>Basic Operation .....</b>	<b>VMB-9</b>
Starting Up the Visual Memory Simulator .....	VMB-9
Loading the System BIOS .....	VMB-10
Loading and Executing Applications .....	VMB-11
MAP File .....	VMB-12
Drag & Drop .....	VMB-12

## **Descriptions of Windows and Panels . . . . . VMB-13**

Main Window .....	VMB-14
Menus .....	VMB-14
Toolbar .....	VMB-17
CPU Register Display Function .....	VMB-18
Execution Control .....	VMB-19
Disassembly Function .....	VMB-20
Visual Memory Image .....	VMB-21
Status Lamp .....	VMB-21
Changing the Size of the Main Window .....	VMB-22
System Console .....	VMB-22
Memory Control Window .....	VMB-23
RAM#0, RAM#1 .....	VMB-24
FLASH#0 .....	VMB-25
XRAM .....	VMB-26
SFR .....	VMB-27
VTRBF .....	VMB-28
Break Control Window .....	VMB-29
Break by Breakpoint Address Comparison .....	VMB-29
Display When an Interrupt Is Received .....	VMB-32
Access Reference Monitor .....	VMB-33
Special Function Register Control Window .....	VMB-34
CPU Control .....	VMB-35
LCD .....	VMB-35
INT Control .....	VMB-36
Timer 0 .....	VMB-36
Timer 1 .....	VMB-37
SIO .....	VMB-37
PORT1 .....	VMB-38
PORT3/7 .....	VMB-38
External INT .....	VMB-39
VMU Special .....	VMB-40
Base Timer .....	VMB-40
LCD Snapshot Window .....	VMB-41
Description of Tool Bar Buttons .....	VMB-41
Display by STAD Checkbox .....	VMB-42
Menus .....	VMB-42
Network Monitor Window .....	VMB-43
Trace Panel .....	VMB-45
Hexadecimal Input Pad .....	VMB-47
Environment Settings Window .....	VMB-49
Settings .....	VMB-49
Work Settings .....	VMB-51

## **Networking . . . . . VMB-55**

## **Related Files . . . . . VMB-57**

System Files .....	VMB-58
Application Files .....	VMB-59

## **Warning Messages . . . . . VMB-61**

# Overview

---

The Visual Memory Simulator is a virtual machine system that simulates the Visual Memory hardware. This system can execute, without any special additional processing, programs that were developed for Visual Memory.

## Features

- The Visual Memory Simulator implements almost all of the hardware in the Visual Memory System through software.
- The status of CPU registers and memory can be displayed during execution.
- Program execution can be traced.
- The status of the Special Function Registers can be displayed.
- The Visual Memory Simulator supports debugging functions such as breakpoints and memory fetch breaks.
- Two Visual Memory Simulators can be connected through a network.

The Visual Memory Simulator is composed of several virtual devices, the central one being a virtual CPU. The virtual CPU is designed as an interpreter. Execution files are fetched and executed one instruction at a time. Because peripheral devices are also implemented in the Virtual Memory Simulator, roughly 100% of a program can be checked. Furthermore, no special programs or hardware are needed in order to load execution files into the Visual Memory Simulator.

Because the Visual Memory Simulator is almost entirely software based, there are differences in the operating speed of the Visual Memory Simulator versus actual Visual Memory. Checks that require the actual speed, such as checks of operation timing and sound, should be performed on the actual hardware. The purpose of the Visual Memory Simulator is to verify the program logic; the use of the Visual Memory Simulator in the development cycle should be differentiated from that of the actual hardware.

## Visual Memory Simulator Operating Environment

**Table 2.1**

CPU	Pentium 150MHz or higher recommended
RAM	At least 32MB recommended
OS	Windows 95
HDD free space	At least 5MB
Colors	At least 256 colors
Resolution	At least 1280 x 1024 recommended

---

**Note:** Because the Visual Memory Simulator is a Windows application, it can basically run on any CPU on which Windows 95 is running. However, the virtual CPU is a type of interpreter, and when the operation of other virtual devices is also included, the Visual Memory Simulator can run slowly. In order to run at a reasonable speed, a PC with a fairly fast clock speed is required.

---

## Checking Operation on Actual Visual Memory Hardware

The Memory Card Utility, which is provided with the Visual Memory SDK, is used to transfer an application that has been developed into Visual Memory. The Memory Card Utility is a utility that is used to transfer Visual Memory applications between a PC and the Dev.Box, and between the Dev.Box and Visual Memory.

The Memory Card Utility is located in the "Utility" folder in the folder where the Visual Memory SDK was installed, as an ELF file that runs on the Dev.Box. For details on how to use the Memory Card Utility, refer to the "Visual Memory Tutorial." After the logic in an application has been checked by using the Visual Memory Simulator, be certain to check the timing and operating speed of the application on the actual hardware.

The following environment is needed in order to run the Memory Card Utility:

### **Items provided by Sega:**

- Dreamcast SDK
- CodeScape (including DA Checker)
- GD WorkShop
- Dev.Box (Set 5.2X or later)
- Dreamcast controller
- Visual Memory

### **Items That Must Be Obtained Separately**

- RS-232C cross cable
- Communications program that runs under Windows

# Notes Concerning Startup for the First Time

When the Visual Memory Simulator is started up for the first time, the contents of flash memory bank 1 are undefined, so the Visual Memory Simulator may indicate that "Visual Memory has not been formatted."

The operation described below must be performed the first time that the Visual Memory Simulator is started up.

---

**Caution:** Be certain to perform the procedure described below when starting up the Visual Memory Simulator for the first time after installing the Visual Memory SDK.

---

⌚ Execute the Visual Memory Simulator.

⌚ From the [File] menu, select [Open FLASH#1 Memory].

The following screen appears:

From the "Files" folder, select "GAME . BIN" and then click the [Open] button.

---

**Caution:** "GAME . BIN" contains the memory image for flash memory bank 1 in Visual Memory. Because this file includes the FAT information and the system management information, if this information is not found in flash memory bank 1, Visual Memory will be recognized as not having yet been formatted.

---

⌚ Once "GAME . BIN" is loaded, the following screen appears:

☞ From the [Option] menu, select [Environment Variables]; the following screen appears:

☞ Make the settings described below in the dialog box that appears. All of these items are displayed under the [Settings] tab.

In the [Start Up] group, click the [Load System File] checkbox so that the box is checked.

In the [System File] group, select the [Quick Start BIOS] option.

After you have made all of the settings, click the [OK] button.

☞ The display returns to the Visual Memory Simulator screen.

From the [File] menu, select [Exit] to quit the Visual Memory Simulator.

---

**Caution:** Be certain to quit the Visual Memory Simulator.

---

When you have completed the above procedure, the Simulator will recognize Visual Memory as having been formatted.



# ***Implemented Devices***

---

The following devices are implemented in the Visual Memory Simulator:

- Virtual CPU
- Memory
- LCD controller
- Serial interface
- Timer
- Interrupt controller
- I/O ports
- External input devices

## **Virtual CPU**

A CPU interpreter, called the "virtual CPU," and which executes the Sanyo Electric LC86 Series instruction set, is implemented in the Visual Memory Simulator. This virtual CPU executes binary code that is stored in the memory area in the same manner as the actual CPU. There is no need to add special programs for the Simulator.

The Windows system idle is used as the operating clock for the virtual CPU. "n" instructions are executed per idle. The number of instructions that are executed per idle can be set in the Environment Settings Window. Adjust this value according to the clock speed of the PC that you will be using. for a detailed description of how to make this setting, refer to [Settings] - [CPU Loop Count] in section, "Environment Settings Window." However, if this value is increased, the timing by which Windows messages are acquired becomes skewed, with the result that the response of buttons, etc., becomes sluggish.

## Memory

The Visual Memory Simulator simulates all memory areas in the Visual Memory System.

### RAM Area

Bank 0	00H to FFH (256 bytes)	System work area
Bank 1	00H to FFH (256 bytes)	User work area

### ROM Area

This area stores the OS program and the system application. This area cannot be manipulated by the user.

### Flash Memory

Bank 0	0000H to FFFFH (64K bytes)	User program area
Bank 1	0000H to FFFFH (64K bytes)	Backup memory area

### Work RAM

This is a buffer for communications with Dreamcast that can be accessed through the Special Function Registers.

If there are no communications with Dreamcast, this area can be used as RAM by an application.

VTRBF	0000H to 01FFH (512 bytes)
-------	----------------------------

### XRAM

This is the LCD display memory. This memory is equivalent to video memory in a typical PC.

This memory consists of three banks. Two banks are allocated to bitmap display, and one bank is allocated for icons.

Bank 0	0180H to 01FBH (96 bytes)	Includes unused areas.
Bank 1	0180H to 01FBH (96 bytes)	Includes unused areas.
Bank 2	0180H to 0185H (6 bytes)	

In addition to direct access by the virtual CPU, these memory areas can be edited through the Memory Control Window.

## LCD Controller (LCDC)

The LCDC is designed to operate in an equivalent manner to the actual hardware from the standpoint of the CPU. The LCDC is accessed through the LCD-related Special Function Registers.

If data is written to XRAM while the LCD is ready for display, the data appears in the drawing area.

## Serial Interface (SIO)

The serial interface is used to connect two Visual Memory units. A Visual Memory unit has two SIOs. SIO0 is allocated for sending, and SIO1 is allocated for receiving. Together, full-duplex communication is implemented through these two interfaces.

In the Simulator, Visual Memory units are connected by using TCP communication for communications between SIOs.

Only the Special Function Registers for the SIOs are visible from the standpoint of the virtual CPU; the network is hidden. Connection control is implemented through the [Network Monitor] command on the [Panels] menu.

## Timer

Visual Memory has three timers. The Simulator supports timer 0, timer 1, and the base timer.

The Simulator supports all interrupts that are generated by the timers.

### Timer Restrictions

The counter function based on external input to timer 0 is not supported in the Simulator. Although pulses can be generated through timer 1 PWM output, no sound is actually output.

### Operating Clock for Timers

As is the case for the virtual CPU, the Windows system idle is used for the clock that is supplied to the timers. As a result, the actual speed at which the timers operate is different from that of the actual hardware.

## Interrupt Controller

The LC86 Series supports interrupts with variable priority levels and nested interrupts. The Visual Memory Simulator simulates interrupt operations in the same manner. There are no restrictions concerning interrupts. Both internal and external interrupts are supported.

## I/O Ports

There are three I/O ports: port 1, port 3, and port 7.

- |        |   |
|--------|---|
| Port 1 | Assigned to SIO and PWM output ports.   |
| Port 3 | The Visual Memory buttons are connected to this port.                             |
| Port 7 | The voltage detection and other VMU detection signals are connected to this port. |

# External Input Devices

### **Buttons Connected to Port 3**

Visual Memory has eight buttons that serve as input devices. These buttons are connected to port 3. The current status (pressed / not pressed) of each button can be detected by reading this port. When a button is not being pressed, the corresponding signal is high; when a button is being pressed, the corresponding signal is low. Port 3 interrupts are also supported, so it is possible to simulate interrupts that are generated when a button status changes.

Starting from the most significant bit, the buttons assigned to the bits are: SLEEP button, MODE button, B button, A button, right button, Left button, Down button, Up button.

### **Control Signals Connected to Port 7**

Bits 0 to 3 of port 7 are input signal ports for external interrupts. Interrupt control for external interrupts is specified through the IO1CR register and the I23CR register.

Four input signals are connected to port 7.

### **+5V Supply Signal as External Power Supply Connected to P70**

When no external power supply is connected, this signal is low; when external power is supplied, this signal is high. This is simulated through the "+5V Test" checkbox connected to P70 on the SFR panel. When "ON," external power is being supplied.

### **Internal Battery Voltage Drop Signal Connected to P71**

This signal is generated when the voltage of the internal battery drops. When this signal is high, the battery voltage is normal. When this signal is low, the battery voltage drops. This is simulated through the "Low Voltage Test" checkbox connected to P71 in the SFR panel. When "ON," the voltage is low.

### **Input Signals ID0, Connected to P72, and ID1, Connected to P73**

These signals are normally low; they are high when an input is connected. This is simulated through the "ID0 Test" and "ID1 Test" checkboxes connected to P72 and P73 in the SFR panel.

# ***Basic Operation***

---

This chapter explains the procedure for loading and executing application programs in the Visual Memory Simulator.

## **Starting Up the Visual Memory Simulator**

Startup the Visual Memory Simulator either from the Windows [Start] menu, or directly from the folder where it was installed. Once the Simulator is started up, the Main Window is displayed and the Simulator begins waiting for input.

## Loading the System BIOS

Right after the Visual Memory Simulator has been started up, the system ROM area is initialized. Because applications developed by users are called from the system BIOS, it is necessary to load the system BIOS first.

- 1) From the Visual Memory Simulator's [File] menu, select [Open System File].
- 2) Select the system BIOS file (SBF) to be loaded.
- 3) Click the [Open] button. The system BIOS is loaded into the internal system ROM.

"FBIOS.SBF" is the full-size BIOS; this program manages the system when Visual Memory is started up. "QBIOS.SBF" is the quick start BIOS; this BIOS can skip the clock setting that is requested when Visual Memory is reset.

Applications are called from BIOS and started up. A setting can be made in the Environment Settings window that automatically loads the system BIOS when the Visual Memory Simulator is started up. For details on making these settings, refer to [Startup Settings] - [Load System File] in section, "Environment Settings Window."

---

**Caution:** Quick start BIOS supports exactly the same functions as full-size BIOS, except that the clock setting can be skipped at startup.

---

## Loading and Executing Applications

The application execution files that can be loaded into the Visual Memory Simulator are HEX files. The extension for such files is ".HEX" or ".H??".

---

**Caution:** The Visual Memory Simulator cannot load binary files (".BIN") created by H2BIN.

---

- 1) From the [File] menu, select [Open Application].
- 2) Select the application execution file (".HEX" or ".H??") to be loaded.
- 3) Click the [Open] button. The file is loaded into flash memory bank 0. The memory area where such files are loaded is fixed to "bank 0."

After the file has been loaded, click the Reset button; the Simulator virtual machine is reset and the CPU begins operating. As soon as the CPU begins operating, the system BIOS is executed.

Although the operations performed in the CPU's internal registers while the system BIOS is running can be checked, displaying the registers consumes CPU time, so the Simulator will run more slowly as a result. The display of these registers can be stopped in order to speed up processing.

To stop an application that is running, click the [Break] button. The values in the registers as of the moment when execution was stopped are displayed on the console, and operation stops. Furthermore, the program counter value for the next instruction to be executed is set in the text box where the execution address is stored.

To resume program execution, click the [Run] button. Click the [Step] button to step through the instructions one at a time.

### **MAP File**

When a MAP file is in the same folder as the application, this file is loaded after the application. The extension for symbol files is ".MAP", and this type of file can be output by the Linker. Although this file is not required, it allows symbol names to be displayed during disassembly.

The symbols that are loaded are stored in list format in the hexadecimal input pad.

---

**Caution:** The extension for files output by the Linker is ".EVA". This type of file is converted to a HEX file by E2H86K.EXE. Because the Visual Memory Simulator cannot load EVA files, these files must be converted to HEX files.

---

### **Drag & Drop**

The drag & drop technique can be used with the text boxes where addresses are input. In order to begin dragging from a given text box, hold down Shift key and press the left mouse button. The mouse cursor changes to a drag cursor, confirming that dragging is enabled.

The address labels and the hexadecimal input pad text boxes that are displayed on the Special Function Register Panel can be dragged without using Shift key. When the mouse is moved to one of these areas, it changes to a drag cursor.



# ***Descriptions of Windows and Panels***

---

When the Visual Memory Simulator is started up, the Main Window is displayed first. If all that is necessary is to load and execute an application program that has been created, the functions in the Main Window are all that are needed. Debugging requires the use of functions on a number of other windows.

## **Main Window**

This is the main window for the Visual Memory Simulator. Applications can be loaded and execution can be controlled through this window.

## **Memory Control Window**

This window displays the contents of memory implemented in Visual Memory. The contents of memory can also be edited on this screen.

## **Break Control Window**

This window is used to set execution stop triggers for breakpoints.

## **Special Function Register Control Window**

This window displays the status of the Special Function Registers.

## **LCD Snapshot Window**

This window gets and enlarges images that are displayed on the LCD.

## **Network Monitor Window**

This control window is used to connect two Visual Memory Simulators.

## **Trace Panel**

This panel is used to perform program traces.

## **Hexadecimal Input Pad**

This window is used to easily input hexadecimal numbers. A symbol table is also stored here.

## **Environment Settings Window**

This window is used to make the basic settings for Visual Memory Simulator operation.

## Main Window

The following functions are implemented in the main window:

- Loading applications and system files
- Calling up control windows and panels
- Displaying CPU registers
- Executing, stopping, and step-executing applications
- Outputting disassembled listings
- Simulating the Visual Memory LCD and buttons
- Switching the Main Window between reduced and normal size display

The following Main Window functions are described in this section:

- Menus
- Speed button
- CPU register display function
- Execution control
- Disassembly function
- Visual Memory image
- Status lamps
- Changing the Main Window size
- System console

## Menus

### File Menu

#### [Open Application] Command

This command loads an application in HEX file format. The application is loaded into flash memory bank 0. Flash memory bank 0 is used as memory for Visual Memory applications.

#### [Re-open Application] Command

This command reloads the application that is currently open. This command cannot be selected initially; it can only be used after an application has been loaded. The name of the file that was loaded is displayed in the title bar on the Main Window.

#### [Open System File] Command

This command loads the system BIOS into the internal ROM area. A setting can be made in the Environment Settings Window that will load the system BIOS automatically when the Visual Memory Simulator is started up.

### [Open RAM File] Command

This command loads a RAM file that was saved. A RAM file contains the contents of RAM that were saved. RAM banks 0 and 1, work RAM, and XRAM are included in a RAM file.

The file format is binary. The memory map is as shown below.

0000H - 00FFH	RAM bank #0
0100H - 017FH	SFR (reserved for system)
0180H - 01FFH	XRAM bank #0
0200H - 027FH	Reserved for system
0280H - 02FFH	XRAM bank #1
0300H - 037FH	Reserved for system
0380H - 03FFH	XRAM bank #2
0400H - 04FFH	RAM bank #1
0500H - 06FFH	VTRBF
0700H - FFFFH	Reserved for system

### [Save RAM File] Command

This command saves the current contents of RAM provided for the virtual CPU. The file format is binary. This type of file can be loaded by using the [Open RAM File] command.

### [Open FLASH#1] Command

This command loads a file into flash memory bank 1. The file format is binary. The size of flash memory bank 1 is 64K. Writing to flash memory is accomplished by writing directly to the memory area, ignoring the flash write simulation facility.

Flash memory bank 1 is a system area that is used to manage Visual Memory files, and an area for saving Dreamcast game data. Visual Memory applications are loaded into flash memory bank 0.

---

**Caution:** If "GAME.BIN" is not loaded through this menu before starting up an application, an error message stating that Visual Memory has not been initialized will be displayed. Before starting up an application, be certain to first load "GAME.BIN" through this menu.

---

### [Save FLASH#1] Command

This command saves the current contents of flash memory bank #1 in a file. The file format is binary. This type of file can be loaded by using the [Open FLASH#1] command.

### [Print] Command

This command prints the character string that is displayed in the text box (system console) at the bottom of the Main Window.

### [Save Console to File] Command

This command saves the character strings displayed in the system console in a text file.

### **[Exit] Command**

This command quits the Visual Memory Simulator.

### **[Execute] Menu**

### **[Break] Command**

This command halts application execution. The effect of this command is identical to that of the Break button.

### **[Reset] Command**

This command resets the virtual Visual Memory, and starts Visual Memory operation.

### **[Run/Continue] Command**

This command executes the program, starting from the instruction following the instruction at which program execution was stopped.

### **[Step Execution] Command**

This command executes one instruction according to the current program counter.

### **[Disassemble] Command**

This command displays a disassembled listing.

### **[Panel] Menu**

### **[Break Control] Command**

This command displays the Break Control Window.

### **[Memory Control] Command**

This command displays the Memory Control Window.

### **[SFR Display] Command**

This command displays the Special Function Register Window.

### **[LCD Snapshot] Command**

This command displays the LCD Snapshot Window.

### **[Network Monitor] Command**

This command displays the Network Monitor Window.

### [Trace Panel] Command

This command displays the Trace Panel.

### [Hexadecimal Input Pad] Command

This command displays the Hexadecimal Input Pad.

### [Reduce Main Window] Command

This command changes the size of the Main Window to the size of the Visual Memory image.

### [Normal Main Window] Command

This command restores the Main Window to its normal size.

### [Options] Menu

#### [Environment Settings] Command

This command displays the Environment Settings Window.

#### [Clear Console] Command

This command clears the system console.

### [Help] Menu

#### [Reference Guide] Command

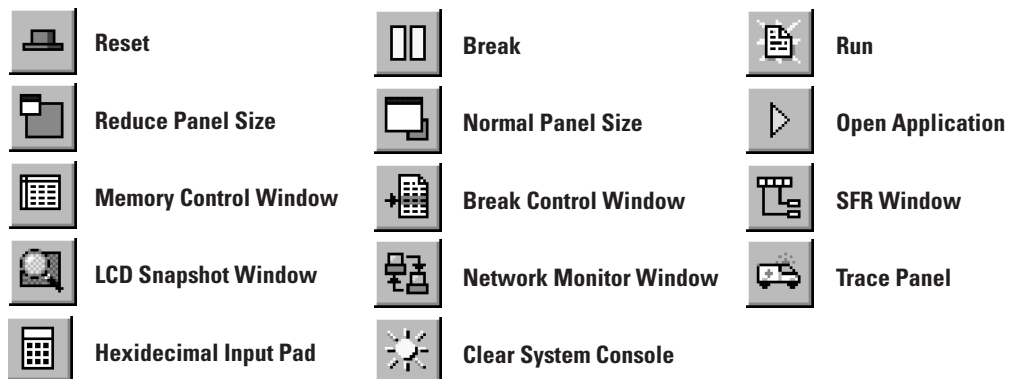
This command displays help.

#### [Version Information] Command

This command displays the version information for the Visual Memory Simulator.

## Toolbar

The toolbar is located at the top of the panel. The toolbar buttons all correspond to menu items or buttons on panels.



## CPU Register Display Function

This function displays the values of the virtual CPU's registers in the Main Window.

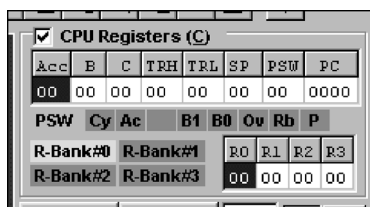


Figure 1.3

The registers are Acc, B, C, TRH, TRL, SP, PSW, and PC. Each value is expressed in hexadecimal notation. Each register may be edited. After selecting the register to be edited by clicking on the register, click on the register again to begin editing it. In the case of the PSW (Program Status Word), the status of each bit is displayed.

### Meanings of the Bits in the PSW

Cy	Carry flag
Ac	Auxiliary carry flag
B1	Indirect register bank specification bit
B0	Indirect register bank specification bit
Ov	Overflow flag
Rb	RAM bank switching bit
P	Parity bit

Each bit of the PSW, except for the parity bit, can be inverted by clicking on the bit with the mouse. The results of the change are reflected in the value of the PSW.

The selected bank and the current indirect register values are displayed in the indirect registers.

The contents of the register scan be edited. It is also possible to change the current bank for the indirect registers by clicking on the label that indicates the bank. If B1 and B0 in the PSW are changed, the bank label is also updated.

Because the CPU registers can be displayed while an application is in progress, so the changes in register values can be observed. However, because it takes time to update the value of each register, the operating speed of an application will slow down if the register values are displayed. To suppress the register display, uncheck the [CPU Registers] checkbox. When this checkbox is in the checked state, the contents of the registers are displayed.

## Execution Control

There are four buttons that are used for execution control.



Figure 1.4

### Execution Control Buttons

#### Reset Button

If the [Reset] button is clicked, all Visual Memory devices are reset.

All of the CPU registers are initialized with "00H," RAM bank 0 is the current RAM bank, and the internal ROM is selected as the program ROM. "0000H" is loaded into the program counter, and then the CPU begins running.

#### Run Button

If the [Run] button is clicked, execution begins, starting from the address that is shown in the execution start address text box (program counter). In this case, the devices are not reset. The [Sys]/[Usr] button indicates whether the program that is currently executing is located in ROM or in flash memory. The [Sys] button indicates ROM, and the [Usr] button indicates flash memory.

#### Break Button

If the [Break] button is clicked while an application is executing, Visual Memory displays the current register values on the console and halts execution. At this point, the value of the program counter, which is the address of the instruction that is to be executed next, is substituted into the execution start address text box. Execution can be resumed if the [Run] button is pressed right after the [Break] button.

#### Step Button

If the [Step] button is clicked while program execution is halted, the next instruction in the program is executed and then execution halts again. This button can be used to execute a program one instruction at a time in a deliberate fashion.





## Visual Memory Image

The Visual Memory image is a virtual target machine that is patterned on Visual Memory.



**Figure 1.7**

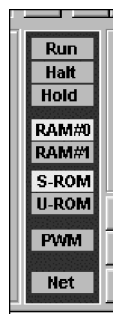
The Visual Memory image includes an area that is equivalent to the LCD, icons, and eight buttons.

The buttons can be clicked through either the mouse or the keyboard. When the Visual Memory image is active, the image is framed in blue. To make the Visual Memory image active, click on any portion of the Visual Memory image other than a button.

The keys that correspond to the Visual Memory buttons can be changed through the environment setting window. For details on how to make this setting, refer to [Work Settings] - [VMU Button Configuration] See “Environment Settings Window” on page 49.

## Status Lamp

There are lamps that indicate the status of Visual Memory located on the right side of the Visual Memory image.



**Figure 1.8**

Run	Lights when the CPU is running. Turns off when the CPU is stopped.
Halt	Lights when the CPU is in the HALT state.
Hold	Lights when the CPU is in the HOLD state.
RAM#0	Lights when RAM bank 0 is selected.
RAM#1	Lights when RAM bank 1 is selected.
S-ROM	Lights when ROM is selected.
U-ROM	Lights when flash memory is selected.
PWM	Lights when PWM is output by an application.
NET	Lights when the Visual Memory unit is connected to another Visual Memory unit.

## Changing the Size of the Main Window

The Main Window can be reduced to a size that displays only the Visual Memory image and the [Reset], [Break], and [Run] buttons on the toolbar, and the panel size change button. In addition, it is possible to make a setting in the Environment Settings Window that sets this reduced size for the Main Window when the Visual Memory Simulator is started up. For details on how to make this setting, refer to [Settings] - [Minimum Size] See “Environment Settings Window” on page 49.



Figure 1.9

## System Console

The system console outputs a variety of information from the Visual Memory Simulator. Text information that is output on the console can be printed or saved in a file.



Figure 1.10

Under the default setting, the console has buffer space for 300 lines. The number of lines can be adjusted in the Environment Settings window. For details on how to make this setting, refer to [Work Settings] - [Console] See “Environment Settings Window” on page 49.

If the text output is longer than the number of lines set for the system console, the text is deleted, starting from the beginning.

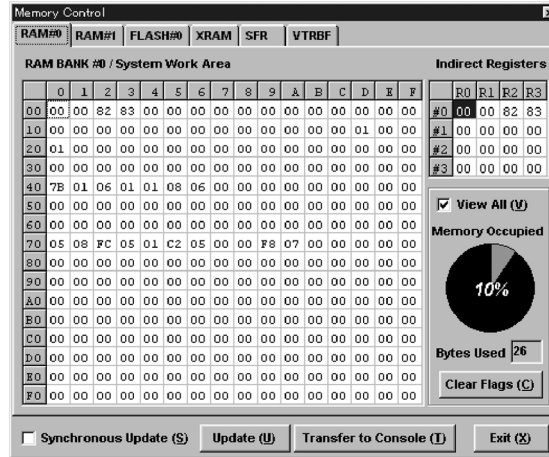
---

**Caution:** When the console buffer is set to a default of 300 lines or more, the operation of the Visual Memory Simulator will slow down.

---

# Memory Control Window

This window displays the contents of memory. Memory is divided into tab pages by category. The memory that is displayed on each page can be the target of editing.



**Figure 1.11**

## Synchronous Display Function

There is a "Synchronous Update" checkbox in the Memory Control Window. If a check is placed in this checkbox, the displayed contents are updated whenever the virtual CPU writes to memory. However, because it takes time to update the screen, the Visual Memory Simulator will run more slowly if this function is used.

## Dump Function

The current contents of the Memory Control Window can be transferred to the system console by clicking the [Transfer to Console] button. The 256 bytes of Flash Memory #0 and work RAM that are currently displayed are transferred to the console.

## Update Button

Clicking the [Update] button causes the Memory Control Window to be updated with the current, most recent data. Normally, this button is used to update the data if the synchronous display function checkbox is not checked.

- RAM#0            System work area for system BIOS, etc. Size: 256 bytes
- RAM#1            Application area. Size: 256 bytes
- FLASH#0        Flash memory area that is used to store user applications.
- XRAM            LCD display memory.
- SFR              Special Function Registers.
- VTRBF            Work RAM. Size: 512 bytes

## RAM#0, RAM#1

RAM is divided into bank 0 and bank 1. Bank 0 is a system work area that is used by the system BIOS. Bank 1 is a work area that is open to user applications.

The Memory Control Window display format is the same for both of these banks. The size of each bank is 256 bytes.

In the LC86 Series CPU, the first 16 bytes of RAM are allocated as the indirect register area. The indirect register area is displayed separately in an easy-to-read format on the panel.

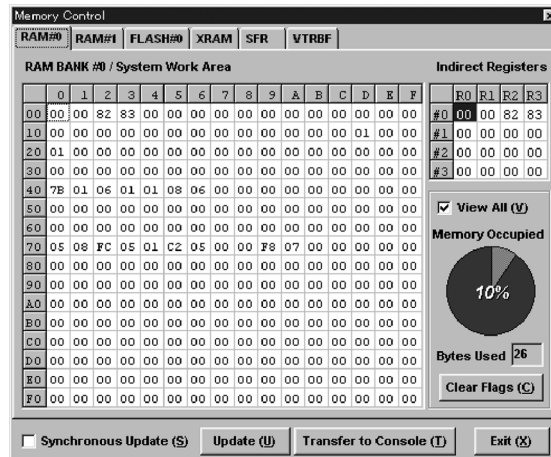


Figure 1.12

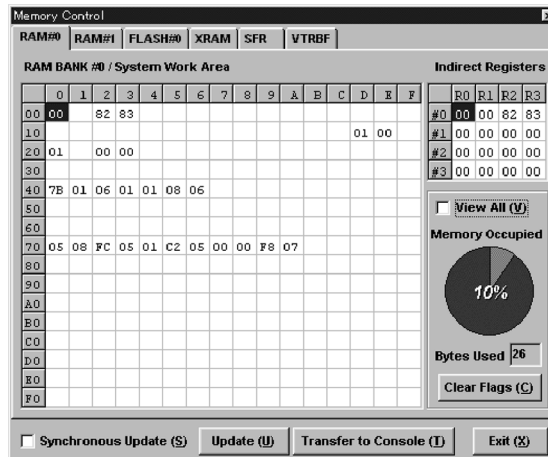
Because RAM bank 0 is allocated for the system BIOS work area and the stack area, it should not be accessed by applications.

RAM bank 1 has 256 bytes that are open to applications.

### Calculation of Memory Usage Rate

RAM contains internal flags that indicate whether a location was accessed by the CPU. When the CPU writes to a given location in memory, the corresponding flag is set. These flags are counted and then used to calculate the memory usage rate. These flags are cleared when the CPU is reset. Specific flags can also be deleted by clicking the [Clear Flags] button.

If the [View All] checkbox is checked, the flags are ignored and all 256 bytes are displayed. If this checkbox is not checked, only memory for which flags have been set is displayed. In other words, the memory that the virtual CPU has written to is displayed.



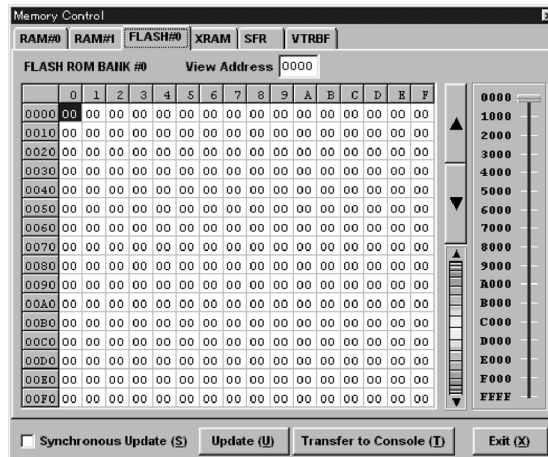
**Figure 1.13**

## FLASH#0

Flash memory is divided into bank 0 and bank 1. The size of each bank is 64 kilobytes each.

Flash memory bank 0 is used for application programs. User-created programs are loaded into this area. Flash memory bank 1 is a data area, so programs cannot be loaded into flash memory bank 1.

**Caution:** The contents of flash memory bank 1 cannot be changed.



**Figure 1.14**

The memory panel displays 256 bytes at one time.



**LCD Bit Image Display**

The current status of XRAM can be displayed as a bit image. This display area can be displayed even when the LCD is off. Although the bit image is synchronized with user writes, it is not synchronized with writes by the virtual CPU. Clicking the [Refresh] button causes the latest contents of XRAM to be displayed.

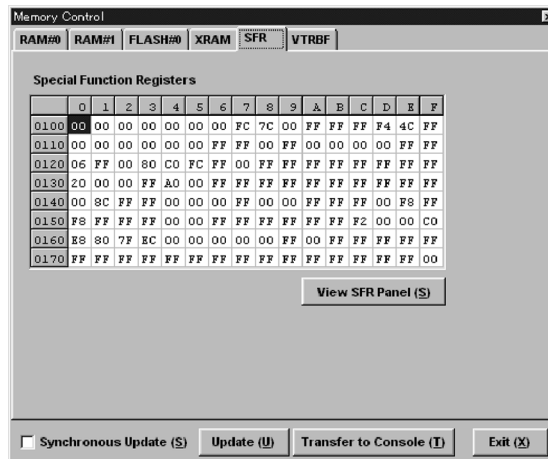
The XRAM display start address can be changed. This specification is made through the Special Function Register STAD. When the [Display By STAD] checkbox is checked, the value in STAD is used as the display start address. If the [Display By STAD] checkbox is not checked, the display starts at the start of XRAM. This is the same as if STAD = 0.

**Note:** The LCD resolution is 48 dots (H) x 32 dots (V), and one line of the LCD corresponds to 6 bytes. The MSB of data that is written corresponds to the left-side dot. XRAM bank 0 is displayed in the top half of the LCD, and bank 1 is displayed in the bottom half. Bank 2 is icon memory.

**Caution:** Because bank 2 is used for the icon that displays the Visual Memory mode, do not change the contents of bank 2 from within an application.

**SFR**

Although the Special Function Registers are displayed, areas that are not actually implemented are also displayed. Normally, locations for which no device is connected are indicated as "0FFH". The data that is displayed can be edited.



**Figure 1.16**

**Caution:** If data is edited in an address that does not exist in any storage that is connected to the device, the data is not actually written.

## VTRBF

VTRBF is allocated as buffer memory for communications between Visual Memory and Dreamcast. If communications with Dreamcast are not being performed, however, this area is open to the user as work RAM. This memory is accessed through the SFRs, and is not decoded in the CPU memory space. The size of this area is 512 bytes, and the addresses range from 0000H to 01FFH.

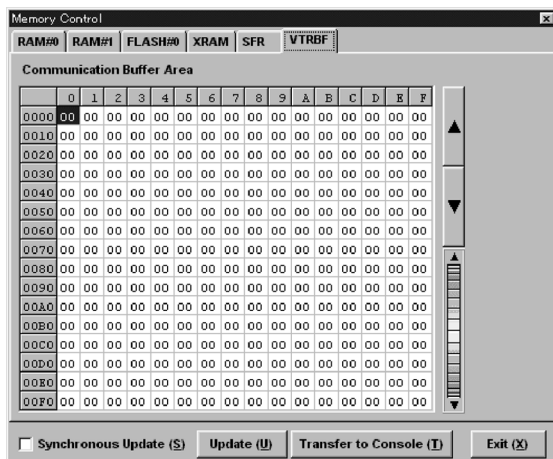


Figure 1.17 Fig. 4-15



## Break Control Window

There are three execution control functions that are implemented in the break control window.

### Break by Breakpoint Address Comparison

Aside from breakpoints, program execution can be stopped by memory fetches.

### Display When an Interrupt Is Received

This indicates that the virtual CPU has received an interrupt and that an interrupt routine has been called.

### Access Reference Monitor

This function displays the position in a program where the specified memory is being accessed.

To enable the address set in the Break Control Window, click the [Apply] button.

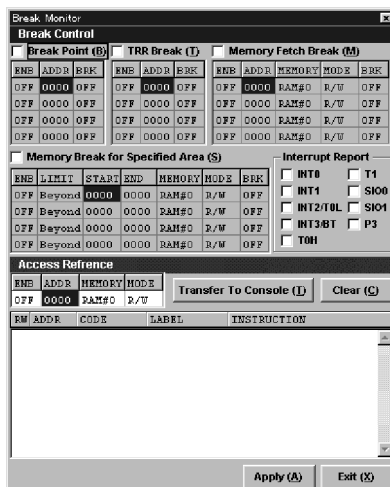


Figure 1.18

## Break by Breakpoint Address Comparison

### Break Control

Break control specifies monitoring of application execution. There are four monitoring groups: breakpoint specification, TRR fetch breaks, memory fetch breaks, and memory fetch breaks with a range specification. Four compare addresses can be set for each group. The common items for all of the groups are explained below.

### Group ON/OFF

This item turns individual groups on and off. The switches for the individual groups are provided in order to minimize address comparison overhead in the Visual Memory Simulator. If a group is turned on, it is displayed on a white background and its settings are enabled.

## Address ON/OFF

Each compare address in a group has its own ON/OFF switch. If set to ON, that compare address is enabled. Addresses can be turned on and off by clicking in the first column.

## Break Mode

The break mode specifies whether to stop or continue execution when a compare address matches. When a compare address matches, the current register values are dumped. If "Break" is ON, the virtual CPU stops executing the program after the register dump. If "Break" is OFF, the register dump is still performed, but the virtual CPU continues executing.

## Breakpoints

Execution is halted when the program counter matches the specified address. Four addresses can be specified as compare addresses.

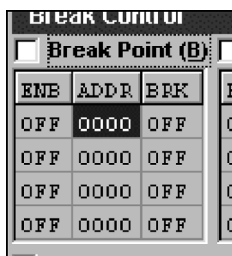


Figure 1.19

## TRR Fetch Break

Execution halts when the program counter matches the memory address that is referenced by the TRH and TRL (indirect address) registers.

Essentially, the program counter is compared with the address that is referenced when the LDC instruction was executed.

Therefore, when the LDC instruction is executed, the address indicated by TRH and TRL is the object of comparison, with no distinction made for flash memory.

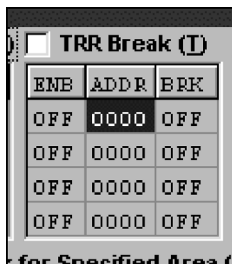


Figure 1.20

**Memory Fetch Break**

A memory fetch break halts execution when the CPU accesses the specified address in memory. This group permits specification of the target memory and the access mode.

The target memory can be RAM#0, RAM#1, SFR, XRAM#0, XRAM#1, or VTRBF. To select the target memory, click on the column that is to be set, and then make the selection in the popup menu that appears. Select the access mode from among READ, WRITE, and R/W.

If the access mode is READ, execution is halted when a read is executed in the target memory; if the access mode is WRITE, execution is halted when a write is executed in the target memory. If the access mode is R/W, execution is halted when a read or a write is executed in the target memory.

<input type="checkbox"/> Memory Fetch Break (M)				
ENB	ADDR	MEMORY	MODE	BRK
OFF	0000	RAM#0	R/W	OFF
OFF	0000	RAM#0	R/W	OFF
OFF	0000	RAM#0	R/W	OFF
OFF	0000	RAM#0	R/W	OFF

Figure 1.21

**Memory Fetch Break With Range Specification**

A memory fetch break with range specification halts execution when an access is made inside or outside of the specified memory address range.

The compare range is specified with a [Start] address and an [End] address. The compare condition can be selected as either "inside the range" or "outside the range." If "outside the range" is specified, then execution stops when memory is accessed outside of the specified address range. This condition does not include the specified addresses. If "inside the range" is specified, then execution stops when memory is accessed inside of the specified address range. This condition does include the specified addresses.

Just as in the case of a memory fetch break, this group permits specification of the target memory and the access mode.

The target memory can be RAM#0, RAM#1, SFR, XRAM#0, XRAM#1, or VTRBF.

To select the target memory, click on the column that is to be set, and then make the selection in the popup menu that appears.

Select the access mode from among READ, WRITE, and R/W.

If the access mode is READ, execution is halted when a read is executed in the target memory.

If the access mode is WRITE, execution is halted when a write is executed in the target memory.

If the access mode is R/W, execution is halted when a read or a write is executed in the target memory.

<input type="checkbox"/> Memory Break for Specified Area (S)						
ENB	LIMIT	START	END	MEMORY	MODE	BRK
OFF	Beyond	0000	0000	RAM#0	R/W	OFF
OFF	Beyond	0000	0000	RAM#0	R/W	OFF
OFF	Beyond	0000	0000	RAM#0	R/W	OFF
OFF	Beyond	0000	0000	RAM#0	R/W	OFF

Figure 1.22

### Display When an Interrupt Is Received

#### Interrupt Report

When the virtual CPU accepts an interrupt, it outputs an acceptance message on the system console. This message is output after the virtual CPU has gotten the interrupt vector. This is valid when the interrupt source checkbox has been checked.

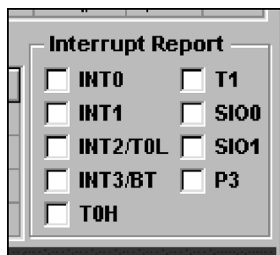


Figure 1.23

These checkboxes will function correctly even if their settings are changed while the CPU is running. The interrupt sources are described below:

INT0	External interrupt. This interrupt is generated when +5V is supplied to the Visual Memory unit.
INT1	External interrupt. This interrupt is generated when the Visual Memory unit's internal battery voltage drops.
INT2/T0L	The external interrupt is generated by ID0, and the internal interrupt is generated by the lower timer 0 register.
INT3/BT	The external interrupt is generated by ID1, and the internal interrupt is generated by the base timer.
T0H	This interrupt is generated by the upper timer 0 register.
T1	This interrupt is generated by timer 1.
SIO0	This interrupt is generated by SIO0.
SIO1	This interrupt is generated by SIO1.
P3	This interrupt is generated by port 3.

## Access Reference Monitor

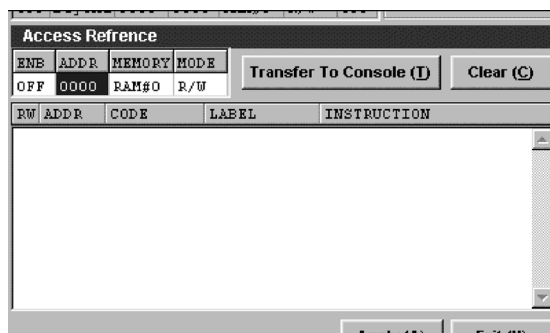
The access reference displays the position in a program that is accessing the specified memory. Usually, this function is used to pinpoint a position in a program that is destroying memory.

The access reference monitor function permits selection of the access mode.

The access mode may be specified as either READ, WRITE, or R/W.

The displayed contents are the mode in which the access was made (R or W), and a disassembly of the program position. This information is output on a special console.

Because program positions are checked twice when output, they are not listed for each access. If you wish to know the access sequence over time, use the memory fetch break function. If you use the access reference monitor, the information will be output on the system console each time an access occurs.



**Figure 1.24**

# Special Function Register Control Window

This window displays the Special Function Registers that Visual Memory leaves open to users.

The display in this window is divided into several groups.

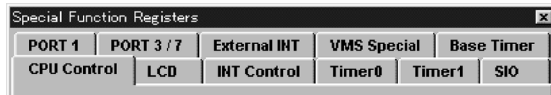


Figure 1.25

Each group is a tabbed page; click on the tab for the group that you want to display.

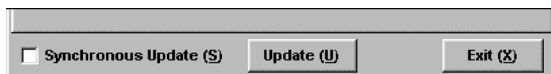


Figure 1.26

Click the [UPDATE] button in order to display the most recent information. Checking the [Synchronous Update] box causes the contents of registers to be updated as soon as the virtual CPU performs a write.

Each register can be edited at the bit level. Clicking on one of the displayed bits causes the value of that bit to be inverted. Bits can also be inverted by clicking on the label connected to that bit.

The Special Function Register groups displayed in this window are listed below.

- CPU Control
- LCD
- INT Control
- Timer0
- Timer1
- SIO
- PORT1
- PORT3/7
- External INT
- VMU Special
- Base Timer

## CPU Control

This group includes the CPU power control, system clock oscillation source control, and external memory control registers. The target registers are PCON, OCR, and EXT.

PCON is the power control register, OCR is the oscillation control register, and EXT is the external memory register.

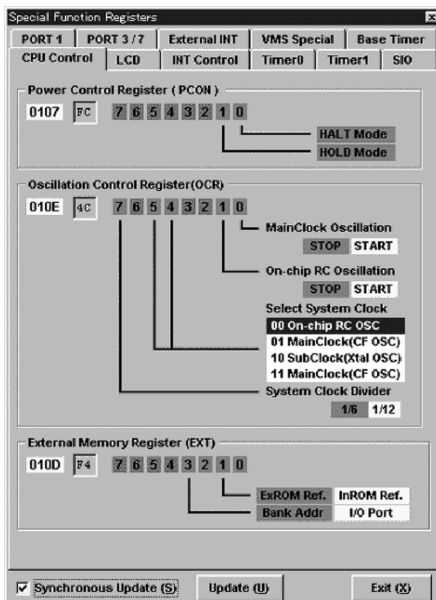


Figure 1.27

## LCD

This group displays the LCD control registers. The target registers are MCR, STAD, CNR, RDR, XBANK, and VCCR. STAD, CNR, TDR, and VCCR cannot be written while the liquid crystal display controller is stopped. This also applies to accesses from an application.

XBANK is unrelated to the operation of the LCD controller, and can be accessed at any time. Although it may appear that it is possible to set this to an unused bank, such a setting is corrected to bank 0.

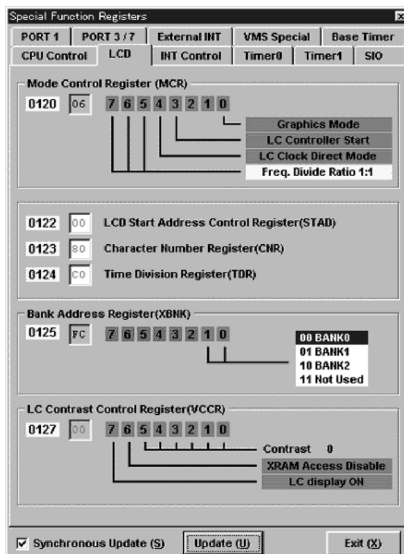


Figure 1.28

## INT Control

This group displays the interrupt-related registers. The target registers are IE and IP.

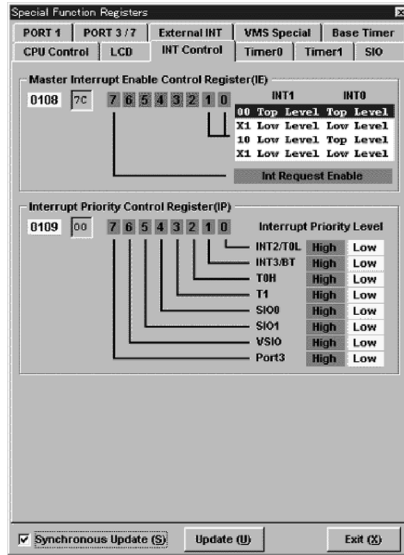


Figure 1.29

## Timer 0

This group displays the registers that are related to timer 0. The target registers are TOCNT, T0PRR, T0L, T0LR, T0H, and T0HR.

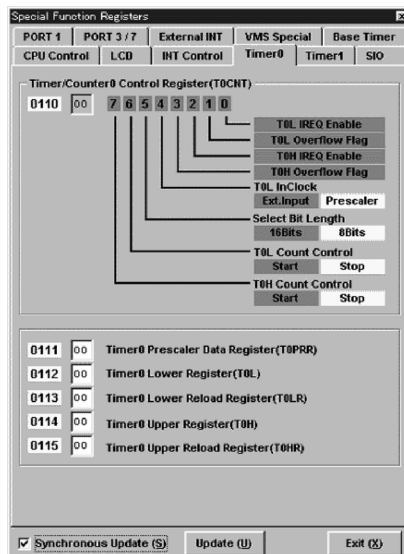


Figure 1.30



## Timer 1

This group displays the registers that are related to timer 1. The target registers are T1CNT, T1LC, T1L, T1HC, and T1H.

The roles of registers T1L and T1H differ, depending on whether they are being read or written. When read, they return the counter value; when written, the value becomes the reload value. Each status can be checked on the SFR panel.

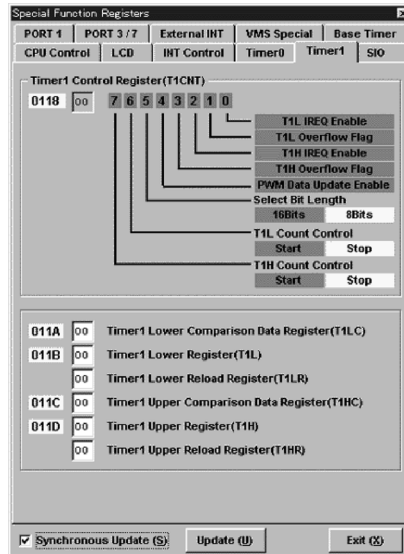


Figure 1.31

## SIO

This group displays the registers for the serial communications-related circuits for two channels. The target registers are SCON0, SCON1, SBUF0, SBR, and SBUF1.

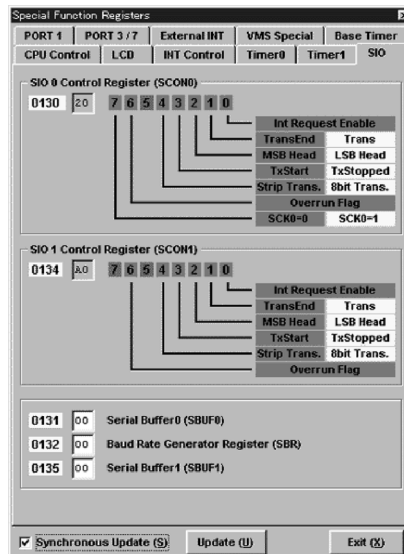


Figure 1.32

## PORT1

This group displays the registers related to port 1. The target registers are P1, P1DDR, and P1FCR.

Because port 1 is used for serial communications and PWM (buzzer) output control, it cannot be used as a general I/O port.

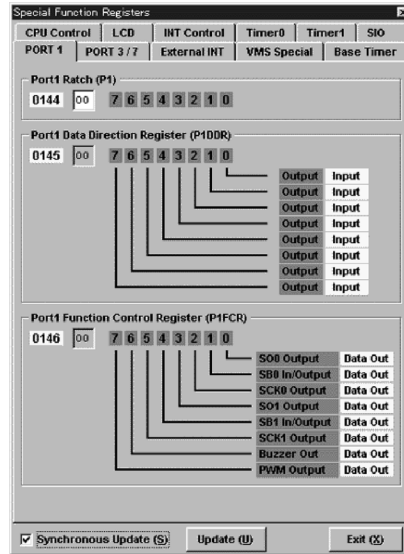


Figure 1.33

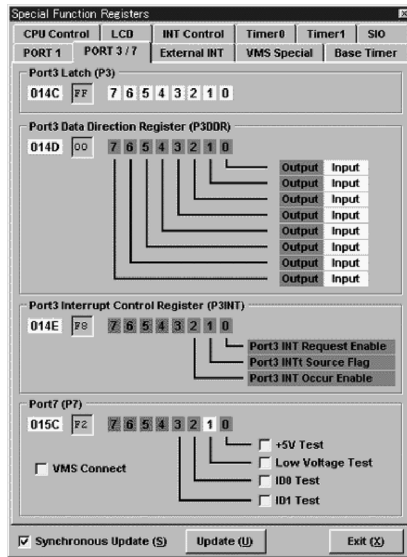
## PORT3/7

This group displays the registers related to port 3 and to port 7. The target registers are P3, P3DDR, P3INT, and P7.

Eight buttons of the Visual Memory unit are connected to port 3. The signals corresponding to each button are normally high, but if a button is pressed the signal goes low. Port 3 interrupts can be generated through P3INT. Note that P3 interrupts are level interrupts.

Port 7 is a four bit input port, with special input signals connected. Each bit of port 7 is an external interrupt input port. Interrupt control is handled through the I01CR and I23CR registers.

- P70 is connected to the +5V supply test checkbox. Normally, this signal is low, but when +5V is supplied this signal goes high. This signal can generate interrupts as external interrupt INT0.
- P71 is connected to the low voltage detection test checkbox. Normally, this signal is high, but when low voltage is detected this signal goes low. This signal can generate interrupts as external interrupt INT1. The signal goes low when the checkbox is checked.
- P72 is connected to the special signal ID0 checkbox. Normally, this signal is low. This signal can generate interrupts as external interrupt INT2.
- P73 is connected to the special signal ID1 checkbox. Normally, this signal is low. This signal can generate interrupts as external interrupt INT3.
- The [VMU Connect] checkbox simulates another Visual Memory unit being connected. When this checkbox is checked, the port values for the connected state are simulated.

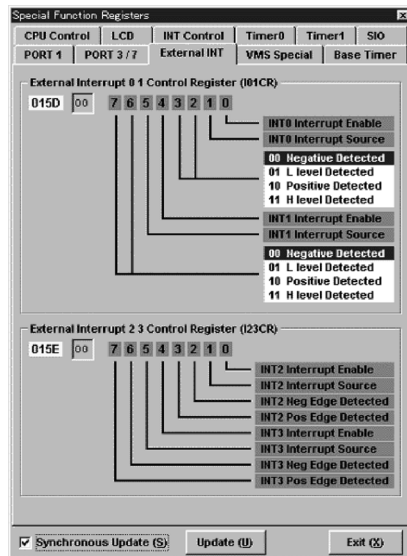


**Figure 1.34**

## External INT

This group displays the external interrupt control-related registers. The target registers are I01CR and I23CR.

INT0 is the +5V supply test, and INT1 is the low voltage detection test. In addition, INT2 is connected to ID0, and INT3 is connected to ID1.



**Figure 1.35**

## VMU Special

This group displays related registers among the registers that are related to the Visual Memory special serial circuitry. The target registers are VCFLG2, VSEL, VRMAD1, VRMAD2, and VTRBF.

VTRBF has [Read] and [Write] buttons. VTRBF can also be listed in the Memory Control Window.

The Visual Memory Simulator only supports registers for access to VTRBF.

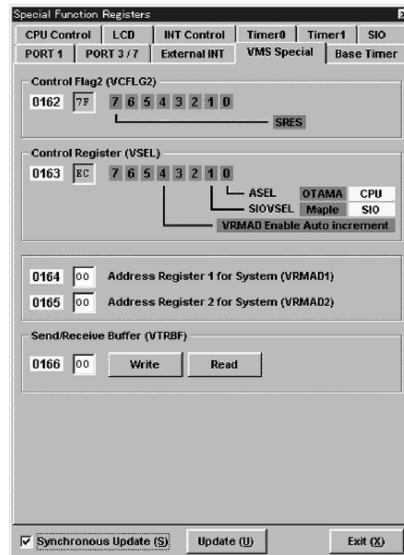


Figure 1.36

## Base Timer

This group displays the registers related to the base timer. The target registers are BTCR and ISL.

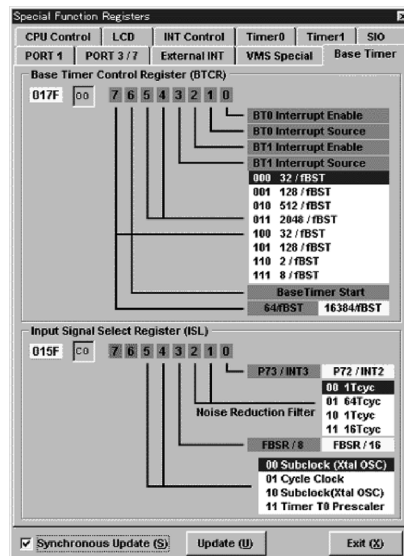


Figure 1.37

## LCD Snapshot Window

This window displays an enlarged version of the bit image that is currently displayed on the LCD. The bit image is fetched either when this window is called or when the [Get Screen] button is clicked. If this window is displayed, the bit image shown is not synchronized with writes by the virtual CPU.

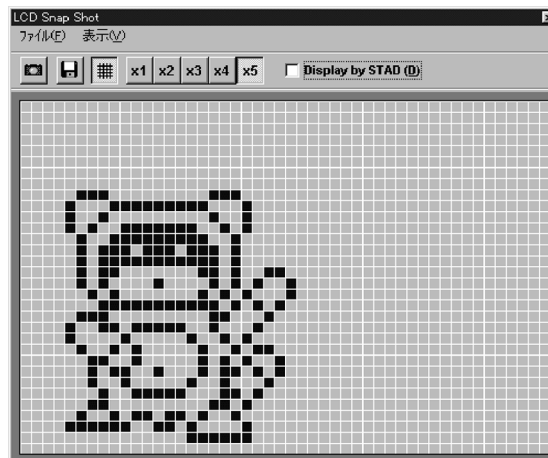


Figure 1.38

### Description of Tool Bar Buttons



**Get Screen Button**

This button gets the current LCD bit image. The contents that are gotten are the contents of XRAM. The dot size is determined by the current magnification.



**Save Button**

This button saves the current bit image in a file with the displayed magnification. The file is saved in ".BMP" format. The grid is not saved.



**Grid Button**

This button displays a grid in the display area. This button functions as a toggle switch; each time the button is clicked, it turns the grid on or off.



**Zoom Button**

This button can be used to select a magnification from 1x to 5x.

### **Display by STAD Checkbox**

The [Display by STAD] checkbox is a switch that enables the display start address register STAD. When this checkbox is checked, drawing is based on addresses converted according to the STAD register. In other words, the same image as that which is displayed on the virtual LCD is displayed on the screen.

If this checkbox is not checked, the STAD register value is ignored when drawing the image. In other words, the contents of XRAM are drawn as is, starting from the beginning of XRAM.

### **Menus**

[File] Menu

[Save Bit Image] command                      Same function as the [Save] button.

[Exit] command                                      Closes the LCD Snapshot Window.

[Display] Menu

[Get Image] command                              Same function as the [Get Screen] button.

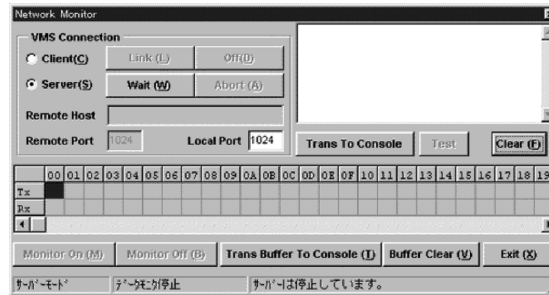
[Display Grid] command                              Same function as the [Grid] button.

## Network Monitor Window

The Visual Memory Simulator simulates data transfers between Visual Memory units via TCP.

The Network Monitor Window supports TCP communications between two Visual Memory Simulators.

This panel consists of several buttons related to connection control, a console for outputting the status, a data monitor for displaying the transferred data, and a status bar for displaying the current status.



**Figure 1.39**

### Connection Control

To perform communications, each unit must select either client mode or server mode. If one is set to server mode, the other must be set to client mode.

### Setting the Unit as the Server

- 1) The option buttons permit selection of either [Client] or [Server]; select [Server].
- 2) In order to set the unit as the server, the local port number must be set. The default setting is 1024.
- 3) If this number is OK, click the [Wait] button. The Visual Memory Simulator is now in server mode in the standby state.

The server performs connection processing when there is a connection request from the client. When the connection is completed, the "Net" lamp in the Main Window lights.

### Stopping Server Operation

Click the [Abort] button to release the server standby state, or to disconnect.

When in the standby state, clicking the [Abort] button puts the unit into the stopped state. If the unit is connected when the [Abort] button is clicked, it performs disconnect processing and then enters the stopped state.

### Setting the Unit as the Client

- 1) Select [Client] with the option buttons.
- 2) Input the machine name or IP address that was set for the server in the [Remote Host] text box.
- 3) Input the port number that was set for the server in the [Remote Port] text box.
- 4) Click the [Link] button.
- 5) Once the connection is made properly with the server, a confirmation message is displayed.

When the connection is made with the server, the "Net" lamp in the Main Window lights.

### **Stopping Client operation**

To release the connection with the server, click the [Off] button. When this button is clicked, the client issues a disconnection request to the server and then enters the unconnected state.

At this point, the server is in standby state. Reconnection is possible by clicking the [Link] button.

### **Console**

The console displays statuses related to connection/disconnection.

The contents displayed in this console can be transferred to the system console by clicking the [Trans To Console] button.

The [Clear] button clears the contents displayed on the console.

Clicking the [Test] button while a connection is established sends a test message to the other side.

### **Data Monitor**

The data monitor monitors the data that is sent between the two Visual Memory Simulators. This data is the data that is transferred from the virtual SIO.

Data that is sent is displayed on the Tx grid, and data that is received is displayed on the Rx grid.

The data monitor function begins operating when the [Monitor On] button is clicked, and stops when the [Monitor Off] button is clicked. When the buffer becomes full, the oldest displayed contents are overwritten first.

The [Trans Buffer to Console] button transfers the currently displayed contents of the data monitor buffer to the system console. The [Buffer Clear] button clears all of the current contents of the buffer.

### **Status Bar**

Starting from the left, the status bar consists of:

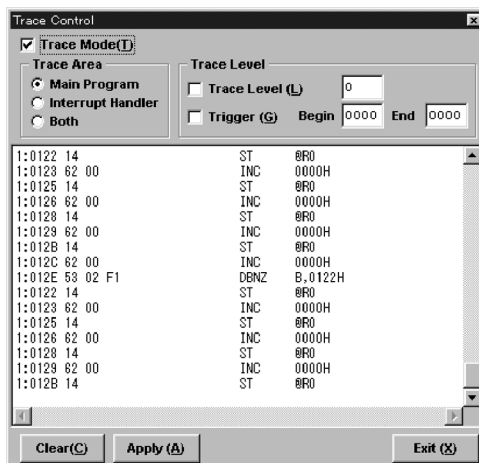
- Client/server mode indication
- Data monitor operating mode indication
- Client, server connection status



## Trace Panel

The Trace Panel traces the execution of an application.

The trace results are output on the trace console.



**Figure 1.40**

### Trace Mode Checkbox

Tracing starts when the [Trace Mode] checkbox is checked. The setting of this checkbox can be changed even while an application is running.

### Trace Area

This specifies the area to be traced.

#### Main Program

This limits tracing to the main program. Here, "main program" indicates areas other than the interrupt processing routine.

#### Interrupt Handler

This traces only the interrupt processing routine (interrupt handler). Tracing starts when an interrupt is received, and continues until the RETI instruction is executed.

#### Both

This traces both the main program and the interrupt processing routine.

### **Trace Level**

The trace level refers to the subroutine nesting level. At level 0, no subroutines are traced. If the level number increases, subroutines to the corresponding nesting level are traced. This setting can be used to avoid unnecessary tracing.

To enable the trace level, check the [Trace Level] checkbox.

### **Trigger**

This is a switch that enables a trace start address and a trace end address.

When the program counter matches the start address, tracing starts and continues until the program counter matches the end address. These addresses are valid only for flash memory.

### **Trace Console**

The trace results are displayed on the trace console. The trace results are the disassembled code that the virtual CPU executed.

To clear the contents of the trace console, click the [Clear] button.

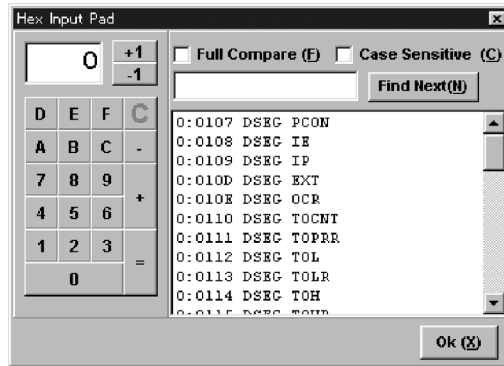
### **Apply Button**

Clicking the [Apply] button places the trace level value, trace start address, and trace end address into effect.

# Hexadecimal Input Pad

## Hexadecimal Input Pad

The Hexadecimal Input Pad is an auxiliary panel that is used to input hexadecimal numbers. Symbols are displayed on the right side of the panel.



**Figure 1.41**

## Description of Input Buttons

### Numeric Buttons

These buttons are used to input hexadecimal digits. The digits that are input are inserted from the right edge, and are then shifted left. Overflow digits are ignored; only the four digits that are displayed are valid.

### C Button

This button clears the displayed digits and returns the display to "0".

### + Button

This button is used for addition in the same manner as a calculator.

### - Button

This button is used for subtraction in the same manner as a calculator.

### = Button

This button displays the total in the same manner as a calculator.

### +1 button

This button adds "1" to the current displayed value. If the current displayed value is "FFFF" and this button is pressed, "0" is the result.

### -1 button

This button subtracts "1" from the current displayed value. If the current displayed value is "0" and this button is pressed, "FFFF" is the result.

### How To Use the Displayed Number

The displayed number can be dragged. When the mouse cursor is moved to the display area, it becomes a drag cursor. The number can be dragged and dropped in an address text box on any panel.

### Keyboard Correspondence

The buttons for the digits also correspond to keys on the keyboard or numeric keypad.

The buttons for the digits "0" through "9", the letters "A" through "F", and the "+" symbol all correspond to the same keys on the keyboard, but the "C" button corresponds to the "\*" key and the "=" button corresponds to the "Enter" key. The "+1" button corresponds to the **PageUp** key, and the "-1" button corresponds to the **PageDown** key.

In order to input from the keyboard, it is necessary to first make the numeric buttons on the Hexadecimal Input Pad active. The numeric buttons can be made active by clicking either on or near the buttons with the mouse.

### Symbol List Box

The symbol information for an application is displayed in the list box. The Special Function Register symbols are registered as the default.

If an appropriate symbol in the list box is selected, that address is transferred to the display box. An address can also be dragged directly from this list box.

### Symbol Search

A symbol search can be performed by inputting the search character in the text box. Each time a character is input, a search is conducted for the symbol that matches that character (incremental search).

The [FindNext] button starts its search from the currently selected position. If the [Full Compare] checkbox is checked, a search is conducted for a symbol that matches the entire character string that was input.

If the [CaseSensitive] checkbox is checked, the search distinguishes between upper and lower case letters.

---

**Caution:** The symbol file is a map file that is output by the Linker. If this file resides in the same folder as the application, it is loaded into the Simulator at the same time as the application. If there is no map file, only the default symbols are available.

---

## Environment Settings Window

The Environment Settings Window is used to make general settings and to make settings concerning the operation of the Simulator.

The items that are set in the Environment Panel are saved in the "VMU.ENV" file in the "Files" folder where the Visual Memory Simulator was installed. This file is loaded when the Visual Memory Simulator is started up, and the environment settings contained in the file are restored.

### Settings

The general settings include the settings upon startup, warning specifications, etc.

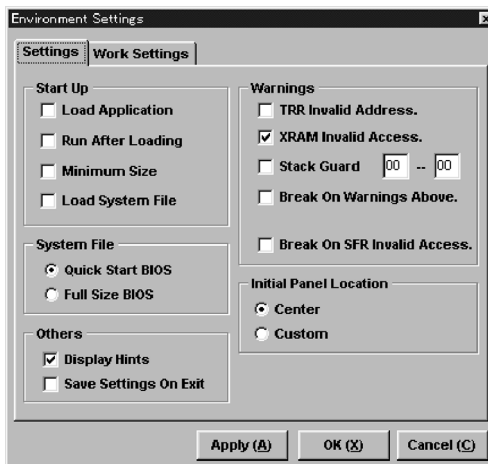


Figure 1.42

#### Startup Settings

##### Load Application

This checkbox is used to automatically load, the next time that the Simulator is started up, the application that is currently loaded. The name of the current application is displayed in the title bar on the Main Window.

##### Run After Loading

This checkbox automatically performs the reset operation and initiates execution the next time that the Simulator is started up. If this checkbox is used at the same time as [Load Application], that application is loaded and then automatically executed.

##### Minimum Size

This displays the Main Window at its minimum size the next time that the Simulator is started up.

##### Load System File

This checkbox automatically loads the system BIOS the next time that the Simulator is started up. The system BIOS file that is loaded is the file that is selected by the system file setting.

**Caution:** This checkbox must be checked in order to execute an application automatically.

---

### System File Setting

This item selects the system file that is loaded by the [Load System File] checkbox. There are two system files: [Quick Start BIOS] and [Full Size BIOS]. Select one or the other by clicking the option buttons.

---

**Caution:** Quick start BIOS supports exactly the same functions as full-size BIOS, except that the clock setting can be skipped at startup.

---

### Warning Specifications

#### TRR Invalid Address

This outputs a warning message when the address that is referenced during the execution of an LDC instruction is outside the application area. "Outside the application area" is defined as an address that is higher than the last address of the HEX file that was loaded.

#### XRAM Invalid Address

This outputs a warning message when an access is made using an XRAM address in a memory area that is not implemented. A warning message is also output when bank 3, which does not exist in XRAM, is specified.

#### Stack Guard

This switch monitors the value of the stack pointer (SP). The monitoring area is specified as a starting and ending value in the text boxes. In the case of an application where the depth of the stack is important, this item can be set in order to output warning messages.

---

**Caution:** If the Visual Memory Simulator is reset, the system BIOS sets "7FH" in SP. Because the data is stored in the stack after the SP is incremented, the actual data is processed starting from 80H, heading up to 0FFH.

---

#### Break On Warning Above

The virtual CPU does not stop program execution when the above warning messages are output. Check this checkbox in order to stop program execution when a warning message is output.

#### Break On SFR Invalid Access

A warning message is always output in the event of an invalid access to the Special Function Registers. Check this checkbox in order to stop program execution when an invalid access is made to the Special Function Registers.

---

### Initial Panel Location

This sets the panel display position. [Center] displays panels in the center of the screen. [Custom] stores the position where the user has moved a panel.

### Others

### Display Hints

This displays hints that have been set up for each GUI control. If this box is checked, hints are displayed; if this box is not checked, hints are not displayed.

### Save Setting On Exit

This specifies whether or not to save application environment information when exiting the Visual Memory Simulator.

## Work Settings

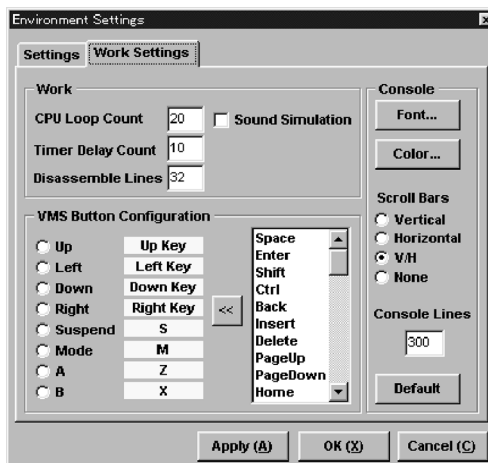


Figure 1.43

### Work Settings

#### CPU Loop Count

This value determines how many instructions the virtual CPU will execute during one system idle process called from Windows. Increasing this value causes the virtual CPU to run faster. If the results of instruction execution are being drawn at the Simulator level, etc., the graphics speed becomes a limiting factor, so that setting a large value for the loop count will have little effect. On the other hand, a large value tends to slow down message handling in windows, with the result that GUI control response becomes sluggish. The operating speed is also affected by the clock speed of the computer on which the Simulator is running, which is another factor that should be taken into account in order to set this value to a suitable level. The default setting is "20."

### **Timer Delay Count**

The timer delay count value is used to adjust the clock to the virtual Visual Memory timer.

The counter for the timer is started after "n" instructions have been executed. "n" is the timer delay count value.

In other words, this value represents the delay before the timer starts operating. If this value is large, the timer slows down. The default setting is "10."

### **Disassemble Lines**

This specifies the number of lines in the disassemble list. This setting is valid when the [Length] checkbox for the Main Window is checked. The default setting is "32."

### **Sound Simulation**

Because the actual hardware needed for PWM output is not available in the Virtual Memory Simulator, the Simulator is not able to output an accurate frequency. When PWM output becomes possible at the Visual Memory Simulator level, the "PWM.WAV" file will be played. This checkbox is used to enable the playback of "WAV" files. When this box is checked, playback is enabled; when this box is not checked, playback is not enabled.

### **VMU Button Configuration**

The settings for the keys that are allocated as the Visual Memory Image buttons can be changed.

Starting from the left, this group consists of option buttons for selecting the Visual Memory buttons, the name of the key that is currently selected, the setting button, and the setting candidate list box.

Select the Visual Memory button that you wish to set from among the option buttons.

Next, select the key to be set from the list box, and then press the setting button [<<]. You can also double-click on the key in the list box. The key that was set is displayed in yellow.

### **Console**

This group sets the font, color, scroll bar, and other options for the system console.

#### **Font Button**

This specifies the character font that is used on the system console. Clicking this button causes the font dialog box to appear. Set whichever font is desired.

---

**Caution:** Although vertical fonts are available in the font dialog box, do not specify any of those fonts.

---



### **Color Button**

This specifies the background color of the system console. Clicking this button causes the color dialog box to appear. Set whichever color is desired.

### **Scroll Bars**

This provides options for the display of scroll bars on the system console.

Vertical	Displays vertical only
Horizontal	Displays horizontal only
V/H	Displays both vertical and horizontal
None	Does not display scroll bars

### **Console Lines**

This specifies the number of lines that are buffered for the system console. The maximum value is 1000 lines. The default setting is 300 lines. Increasing the number of lines increases the load caused by scrolling.

### **Default Button**

This button returns the system console settings to their default settings.



# ***Networking***

---

Two Visual Memory units can be connected to each other through their serial interfaces (SIO). With the Visual Memory Simulator, an equivalent setup can be created by connecting two Simulators through TCP communications.

Although only the various SIO registers are visible from the virtual CPU, data can be transferred to SIO of the other Visual Memory Simulator through the network in response to a transfer request.

The network is controlled through the Network Monitor Window. One of the Visual Memory Simulators is designated as the client, and the other as the server. Because the network connection is not established automatically, it must be established beforehand by using the Network Monitor Window.

Both a client and a server are required for connection. It does not matter which Visual Memory Simulator is the client and which is the server, but it is not possible to have a connection between two clients or two servers.

## **Start up two Visual Memory Simulators in one PC.**

In the Network Monitor Window of the Simulator that will be the client, enter the name of the PC or the IP address as the name of the remote host. Put the server Simulator into the standby state, and then make the connection from the client side.

## **Start up separate Visual Memory Simulators in different PCs.**

Set one of the PCs as the server, and put the Simulator into the standby state. On the client side, enter the name of the server PC or the IP address as the name of the remote host, and then make the connection.

## **Disconnecting the Network**

Although the client and the server can both request disconnection, the disconnection request is usually issued by the client.



## ***Related Files***

---

This section describes the files that the Visual Memory Simulator references.

## System Files

The system files that the Visual Memory Simulator references reside in the "Files" folder.

### **VMU.INI**

This file contains the initial settings for the Visual Memory Simulator. Modifying this file could cause the Visual Memory Simulator to operate incorrectly. The Visual Memory Simulator cannot start up without this file.

### **VMU.ENV**

This file contains the environment settings that have been made by the user. This file is updated when the Visual Memory Simulator is exited.

### **DEFAULT.ENV**

This file contains initial settings for applications to reference if they do not have their own environment file.

### **FBIOS.SBF**

This file contains the ROM image of the system BIOS that is stored into the Visual Memory unit. The system BIOS is started up whenever Visual Memory is reset. Applications are called from the system BIOS, and when an application is exited, control returns to the system BIOS. The system BIOS includes various subroutine packages that can be used by applications.

### **QBIOS.SBF**

QBIOS skips the clock setting screen that is displayed when FBIOS starts up. Because the clock setting can be skipped, program verification can be performed immediately during debugging. In all other respects, QBIOS provides exactly the same functions as FBIOS.

### **PWM.WAV**

Because the Visual Memory Simulator cannot guarantee complete real-time operation, PWM sound output is not possible. When PWM output becomes possible at the Simulator level, the PWM.WAV file will be played.

## Application Files

The files that are referenced by applications are described below.

### **APPFILENAME.H00**

This is the application execution file. The files that the Visual Memory Simulator can load as applications are H00 files. An H00 file is created by converting an EVA file that is output by the Linker.

E2H86K.EXE is used to convert EVA files to H00 files. Although E2H86K.EXE outputs both a HEX file and an H00 file, only the H00 file is used by the Visual Memory Simulator.

### **APPFILENAME.MAP**

The MAP file contains the symbols that are output by the Linker. The file format is that of a typical text file. The Visual Memory Simulator loads this file and extracts the necessary symbols. Once symbols are loaded, they can be displayed with labels when disassembled.

The MAP file is loaded automatically after the application is loaded. Therefore, the MAP file must reside in the same folder as the application. However, the MAP file is not required by the Visual Memory Simulator, so its absence has no effect on the Visual Memory Simulator.

### **APPFILENAME.ENV**

Information such as panel positions and settings can be stored for each application in a file with the "ENV" extension that resides in the same folder as the application. The next time that the application is loaded, this file is referenced and the settings are restored. If this file does not reside in the same folder as the application, the execution of the application is unaffected, except that the default settings will be used.





# ***Warning Messages***

---

This section describes the warning messages that are displayed when an application is executed.

**Stack Guard> Stack overflow occurred.**

If the Stack Guard function has been enabled in the Environment panel, this message appears when the stack pointer has gone outside of the specified range.

**SFR> Invalid write (read) of SFR was attempted.**

This message appears when an invalid write (read) of a Special Function Register was attempted.

An "invalid access" means that a Special Function Register was accessed by a method that is not permitted for users. For example, this message appears in cases where bit access is permitted but byte access is not, or in cases where reading is permitted but writing is not.

**TRR> An invalid address was accessed.**

This message appears when an address referenced by an LDC instruction was outside the range of addresses where the application is loaded.

**XRAM> XRAM cannot be written in subclock mode.**

This message appears when a memory area for which XRAM is not implemented was accessed. The XRAM space exists from 180H to 1FFH, but that does not mean that memory is implemented for that entire area. Addresses in which the lower four bits range from 0CH to 0FH are not implemented.

However, the memory that is implemented in XRAM bank 2 is from 180H to 185H.

### **LCD> Invalid XRAM bank was accessed.**

The XRAM bank specification is made in the XBNK register. The bank number is specified by two binary digits, but the value for bank 3 (which does not exist) can be written to this register. In this case, the Simulator switches the bank to bank 0 and displays this message.

### **LCD> Invalid STAD value was specified.**

This message appears when a value that cannot be set is written in the display address start register for the LCD.

### **SIO#0> Warning: PORT#1 is not ready.**

### **SIO#1> Warning: PORT#1 is not ready.**

SIO uses port 1 for input/output. This message appears when none of the bits in port 1 are set for SIO.

### **SIO> SIO control register values do not match.**

This message appears when the settings in the control registers (SCON0 and SCON1) for two Visual Memory Simulators that are attempting SIO communications do not match, making simulation of SIO communications impossible.

Correct the program so that the settings for the transfer bit length, the LSB/MSB first selection, etc., match for both Visual Memory Simulators.