

**mitsubishi 16-BIT SINGLE-CHIP MICROCOMPUTER
M16C FAMILY**

M16C/80

SERIES

<Assembler language>

Programming Manual

Keep safety first in your circuit designs!

- Mitsubishi Electric Corporation puts the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage. Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of non-flammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

- These materials are intended as a reference to assist our customers in the selection of the Mitsubishi semiconductor product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Mitsubishi Electric Corporation or a third party.
- Mitsubishi Electric Corporation assumes no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts or circuit application examples contained in these materials.
- All information contained in these materials, including product data, diagrams and charts, represent information on products at the time of publication of these materials, and are subject to change by Mitsubishi Electric Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Mitsubishi Electric Corporation or an authorized Mitsubishi Semiconductor product distributor for the latest product information before purchasing a product listed herein.
The information described here may contain technical inaccuracies or typographical errors. Mitsubishi Electric Corporation assumes no responsibility for any damage, liability or other loss rising from these inaccuracies or errors.
- Mitsubishi Electric Corporation semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Mitsubishi Electric Corporation or an authorized Mitsubishi Semiconductor product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
- The prior written approval of Mitsubishi Electric Corporation is necessary to reprint or reproduce in whole or in part these materials.
- If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination. Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
- Please contact Mitsubishi Electric Corporation or an authorized Mitsubishi Semiconductor product distributor for further details on these materials or the products contained therein.

Preface

This manual describes the basic knowledge of application program development for the M16C/80 series of Mitsubishi CMOS 16-bit microcomputers. The programming language used in this manual is the assembly language.

If you are using the M16C/80 series for the first time, refer to Chapter 1, "Overview of M16C/80 Series". If you want to know the CPU architecture and instructions, refer to Chapter 2, "CPU Programming Model" or if you want to know the directive commands of the assembler, refer to Chapter 3, "Functions of Assembler". If you want to know practical techniques, refer to Chapter 4, "Programming Style".

The instruction set of the M16C/80 series is detailed in "M16C/80 Series Software Manual". Refer to this manual when the knowledge of the instruction set is required.

For information about the hardware of each type of microcomputer in the M16C/80 series, refer to the user's manual supplied with your microcomputer. For details about the development support tools, refer to the user's manual of each tool.

Chapter 1 Overview of M16C/80 Series	1
Chapter 2 CPU Programming Model	2
Chapter 3 Functions of Assembler	3
Chapter 4 Programming Style	4
Appendix	Appendix

Guide to Using This Manual

This manual is an assembly language programming manual for the M16C/80 series. This manual can be used in common for all types of microcomputers built the M16C/80 series CPU core.

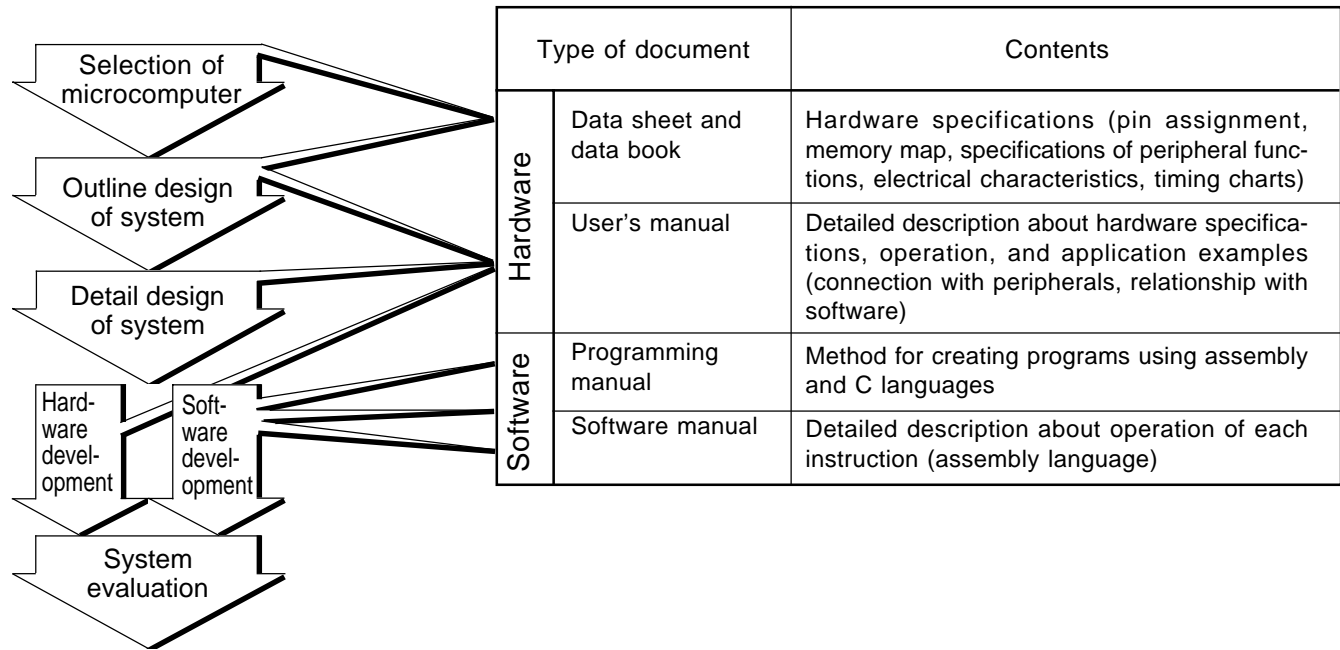
This manual is written assuming that the reader has a basic knowledge of electrical circuits, logic circuits, and microcomputers.

This manual consists of four chapters. The following provides a brief guide to the desired chapters and sections.

- To see the overview and features of the M16C/80 series
→ Chapter 1 Overview of M16C/80 Series
- To understand the address space, register structure, and addressing and other knowledge required for programming
→ Chapter 2 CPU Programming Model
- To know the functions of instructions, the method for writing instructions, and the usable addressing modes
→ Chapter 2 CPU Programming Model, 2.6 Instruction Set
- To know how to use interrupts
→ Chapter 2 CPU Programming Model, 2.7 Outline of Interrupt
→ Chapter 4 Programming Style, 4.3 Setting when using Interrupts
- To check the functions of and the method for writing directive commands
→ Chapter 3 Functions of Assembler, 3.2 Method for Writing Source Program
- To know the M16C/80 series' programming techniques
→ Chapter 4 Programming Style, 4.5 A Little Tips...(Programing technique)
- To know the M16C/80 series' development procedures
→ Appendix Command input form and command parameters in AS308 system

M16C Family-related document list

Usages
(Microcomputer development flow)



M16C Family Line-up

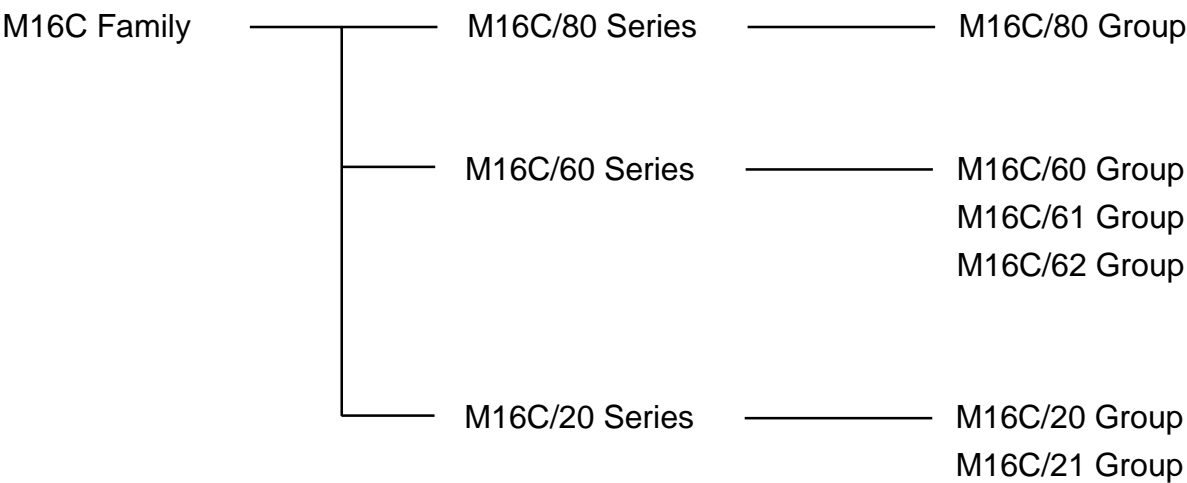


Table of contents

Chapter 1 Overview of M16C/80 Series

1.1 Features of M16C/80 Series	2
1.2 Outline of M16C/80 Group	3

Chapter 2 CPU Programming Model

2.1 Address Space	8
2.1.1 Operation Modes and Memory Mapping	8
2.1.2 SFR Area	10
2.1.3 Fixed Vector Area	15
2.2 Register Set	16
2.3 Data Types	22
Figure 2.3.1 Integer data	22
Figure 2.3.2 Decimal data	22
Figure 2.3.3 String data	23
Figure 2.3.4 Specification of register bits	23
Figure 2.3.5 Specification of memory bits	23
2.4 Data Arrangement	24
2.5 Addressing Modes	25
2.5.1 General Instruction Addressing	27
2.5.2 Indirect instruction Addressing	35
2.5.3 Special Instruction Addressing	38
2.5.4 Bit Instruction Addressing	40
2.6 Instruction Set	45
2.6.1 Instruction Description	46
2.6.2 Instruction List	48
2.6.3 Transfer Instructions	68
2.6.4 Arithmetic Instructions	72
2.6.5 Branch Instructions	80
2.6.6 Bit Instructions	85
2.6.7 Sign-extension instruction	87
2.6.8 Index instruction	88
2.6.9 High-level language and OS support instructions	90
2.7 Outline of Interrupt	95
2.7.1 Interrupt Sources and Vector addresses	95

2.7.2 Variable vector table	98
2.7.3 Interrupt generation conditions and interrupt control register bit configuration	99
2.7.4 Interrupt acceptance timing and sequence	101
2.7.5 Interrupt priority	103

Chapter 3 Functions of Assembler

3.1 Outline of AS308 System	106
3.2 Method for Writing Source Program	109
3.2.1 Basic Rules	109
3.2.2 Address Control	117
3.2.3 Directive Commands	124
3.2.4 Macro Functions	131
3.2.5 Differences with M16C/60	138

Chapter 4 Programming Style

4.1 Hardware Definition	144
4.1.1 Defining SFR Area	144
4.1.2 Allocating RAM Data Area	147
4.1.3 Allocating ROM Data Area	148
4.1.4 Defining a Section	149
4.1.5 Sample Program List 1 (Initial Setting 1)	151
4.2 Initial Setting the CPU	154
4.2.1 Setting CPU Internal Registers	154
4.2.2 Setting Stack Pointer	155
4.2.3 Setting Base Registers (SB, FB)	155
4.2.4 Setting fixed interrupt vector (reset vector)	156
4.2.5 Setting internal peripheral functions	156
4.2.6 Sample Program List 2 (Initial Setting 2)	161
4.3 Setting when using Interrupts	164
4.3.1 Setting Interrupt Table Register(INTB)	164
4.3.2 Setting Variable/Fixed Vectors	165
4.3.3 Setting Interrupt Control Register	166
4.3.4 Enabling Interrupt Enable Flag(I flag)	166
4.3.5 Saving and Restoring Registers in Interrupt Handler Routine	167
4.3.6 Sample Program List 3 (Using interrupts)	170
4.3.7 ISP and USP	174

4.3.8 Multiple Interrupts	177
4.3.9 High-speed interrupts.....	178
4.4 Dividing Source File	182
4.4.1 Concept of Sections	182
4.4.2 Example of program description in divided files	184
4.4.3 Using library files	190
4.5 A Little Tips...(Programing technique)	192
4.5.1 Setup Values of SB and FB Registers	192
4.5.2 Specifying ROM/RAM data alignments	196
4.5.3 Setting stack pointer	198
4.5.4 Using special pages	201
4.5.5 Example for using software interrupt (INTO instruction)	203
4.5.6 Software runaway prevention	205
4.5.7 Method for using the "-LOC" option	209
4.6 Standard processing program	210

Appendix Command input form and command parameters in AS308 system

Appendix A Generating Object Files	Appendix-2
Appendix A-1 Assembling (as308)	Appendix-3
Appendix A-2 Linking(ln308)	Appendix-8
Appendix A-3 Generating Machine Language File (lmc308)	Appendix-12

Table of contents for figure

Chapter 1 Overview of M16C/80 Series

1.1 Features of M16C/80 Series

1.2 Outline of M16C/80 Group

Figure 1.2.1 Block diagram of the M16C/80 group	3
---	---

Chapter 2 CPU Programming Model

2.1 Address Space

Figure 2.1.1 Address space	8
Figure 2.1.2 Operation modes and memory mapping	9
Figure 2.1.3 Control register allocation 1	10
Figure 2.1.4 Control register allocation 2	11
Figure 2.1.5 Control register allocation 3	12
Figure 2.1.6 Control register allocation 4	13
Figure 2.1.7 Processor mode register 0	14
Figure 2.1.8 Memory mapping in fixed vector area	15

2.2 Register Set

Figure 2.2.1 Register structure	18
Figure 2.2.2 Bit configuration of flag register (FLG)	20

2.3 Data Types

Figure 2.3.1 Integer data	22
Figure 2.3.2 Decimal data	22
Figure 2.3.3 String data	23
Figure 2.3.4 Specification of register bits	23
Figure 2.3.5 Specification of memory bits	23

2.4 Data Arrangement

Figure 2.4.1 Data arrangement in register	24
Figure 2.4.2 Data arrangement in memory	24

2.5 Addressing Modes

Figure 2.5.1 Absolute addressing	27
Figure 2.5.2 Address register indirect addressing	28
Figure 2.5.3 Address register relative addressing 1	29
Figure 2.5.4 Address register relative addressing 2	29
Figure 2.5.5 Address register relative addressing 3	29
Figure 2.5.6 SB relative addressing	30
Figure 2.5.7 FB relative addressing 1	30
Figure 2.5.8 FB relative addressing 2	30

Figure 2.5.9 SB relative and FB relative addressing	31
Figure 2.5.10 Application example of SB relative addressing	32
Figure 2.5.11 Application example of FB relative addressing	32
Figure 2.5.12 SP relative addressing 1	33
Figure 2.5.13 SP relative addressing 2	33
Figure 2.5.14 Absolute indirect addressing	35
Figure 2.5.15 Two-stage address register indirect addressing	35
Figure 2.5.16 Address register relative indirect addressing	36
Figure 2.5.17 SB relative indirect addressing	36
Figure 2.5.18 FB relative indirect addressing 1	37
Figure 2.5.19 FB relative indirect addressing 2	37
Figure 2.5.20 Control register direct addressing	38
Figure 2.5.21 Program counter relative addressing 1	38
Figure 2.5.22 Program counter relative addressing 2	38
Figure 2.5.23 Program counter relative addressing 3	39
Figure 2.5.24 Bit instruction absolute addressing 1	40
Figure 2.5.25 Bit instruction absolute addressing 2	40
Figure 2.5.26 Bit instruction register direct addressing	41
Figure 2.5.27 Bit instruction FLG direct addressing	41
Figure 2.5.28 Bit instruction address register indirect addressing	42
Figure 2.5.29 Bit instruction address register relative addressing	42
Figure 2.5.30 Bit instruction SB relative addressing	43
Figure 2.5.31 Bit instruction FB relative addressing	44

2.6 Instruction Set

Figure 2.6.1 Format of instruction description	46
Figure 2.6.2 Specifiers	46
Figure 2.6.3 Typical operations of conditional store instructions	69
Figure 2.6.4 Setting registers for string instructions	70
Figure 2.6.5 Typical operations of string instructions 1	70
Figure 2.6.6 Typical operations of string instructions 2	71
Figure 2.6.7 Typical operations of multiply instructions	72
Figure 2.6.8 Typical operations of divide instructions	73

Chapter 3 Functions of Assembler

3.1 Outline of AS308 System

Figure 3.1.1 Outline of assemble processing performed by AS308	107
--	-----

3.2 Method for Writing Source Program

Figure 3.2.1 Range of sections in AS308 system	117
Figure 3.2.2 Example of address control	120
Figure 3.2.3 Reading include file into source program	121

Figure 3.2.4 Relationship of labels	123
Figure 3.2.5 Example 1 of macro definition and macro call	132
Figure 3.2.6 Example 2 of macro definition and macro call	132
Figure 3.2.7 Example 3 of macro definition and macro call	133
Figure 3.2.8 Example of .LEN statement	136
Figure 3.2.9 Example of .INSTR statement	136
Figure 3.2.10 Example of .SUBSTR statement	137

Chapter 4 Programming Style

4.1 Hardware Definition

Figure 4.1.1 Example of SFR area definition by ".EQU"	144
Figure 4.1.2 Example of SFR area definition by ".BLKB"	145
Figure 4.1.3 Example for setting up a work area	147
Figure 4.1.4 Example for setting a data table	148
Figure 4.1.5 Example for retrieving data table	148
Figure 4.1.6 Example for setting up sections	149
Figure 4.1.7 Description example 1 for initial setting	153

4.2 Initial Setting the CPU

Figure 4.2.1 Example for setting the processor mode and system clock	154
Figure 4.2.2 Example for setting function select registers	155
Figure 4.2.3 Example for initial setting a work area	156
Figure 4.2.4 Example for initial setting ports	157
Figure 4.2.5 Example for setting timer	157
Figure 4.2.6 DMAC-related registers	158
Figure 4.2.7 Example 1 for setting the DMA controller	159
Figure 4.2.8 Example 1 for setting the DMA controller	160
Figure 4.2.9 Description example 2 for initial setting	163

4.3 Setting when using Interrupts

Figure 4.3.1 Variable vector table	165
Figure 4.3.2 Saving and restoring registers in interrupt handling	168
Figure 4.3.3 Saving and restoring registers by register bank switchover	169
Figure 4.3.4 Sample program 3(Using interrupt)	173
Figure 4.3.5 Interrupt number assignments	174
Figure 4.3.6 When a peripheral I/O interrupt or an INT instruction interrupt using software interrupt numbers 0 through 31 occurs	175
Figure 4.3.7 When an INT instruction interrupt using software interrupt numbers 32 through 63 occurs	176
Figure 4.3.8 Execution example of multiple interrupts	177

Figure 4.3.9 Operation of a high-speed interrupt	178
Figure 4.3.10 Program example when using a high-speed interrupt	181
4.4 Dividing Source File	
Figure 4.4.1 Example of sections located in memory	183
Figure 4.4.2 Divided file 1 (WORK.A30)	185
Figure 4.4.3 Divided file 2 (MAIN.A30)	186
Figure 4.4.4 Divided file 3 (SUB_1.A30)	187
Figure 4.4.5 Example of include file	188
Figure 4.4.6 Utilization of directive command .LIST	189
Figure 4.4.7 Creating a library file	190
Figure 4.4.8 Example for linking library files and relocatable module file	191
4.5 A Little Tips...(Programing technique)	
Figure 4.5.1 Example for setting SB and FB registers as having fixed values	192
Figure 4.5.2 When using SB and FB registers dynamically	193
Figure 4.5.3 Program example for using SB and FB registers dynamically	195
Figure 4.5.4 Example of alignment specification	197
Figure 4.5.5 Stack operation and status when an interrupt is accepted	199
Figure 4.5.6 Stack operation and status when a subroutine is called	200
Figure 4.5.7 Example for using special page subroutine call	202
Figure 4.5.8 Example for using INTO (software interrupt) instruction	204
Figure 4.5.9 Operation flow when program runaway is detected	205
Figure 4.5.10 Example of runaway detection program 1	207
Figure 4.5.11 Example of runaway detection program 2	207
Figure 4.5.12 Runaway detection using software interrupt instructions	208
Figure 4.5.13 Example for specifying section data location with -LOC option	209
4.6 Standard processing program	
Figure 4.6.1 Sample program for conditional branching based on specified bit status	210
Figure 4.6.2 Sample program for table retrieval	210
Figure 4.6.3 Example for subroutine call by table jump	211

Appendix Command input form and command parameters in AS308 system

Appendix A Generating Object Files

Figure A.1 Outline of processing by AS308	Appendix-2
Figure A.2 Example of assembler list file	Appendix-6
Figure A.3 Example of assembler error tag file	Appendix-7
Figure A.4 Example of link error tag file	Appendix-10
Figure A.5 Example of map file	Appendix-11

Table of contents for table

Chapter 1 Overview of M16C/80 Series

1.1 Features of M16C/80 Series	
1.2 Outline of M16C/80 Group	
Table 1.2.1 Outline Specifications of M16C/80 Group	4
Table 1.2.2 Register Structure of M16C/80 Series	5

Chapter 2 CPU Programming Model

2.1 Address Space	
2.2 Register Set	
Table 2.2.1 Register Status after Reset Cleared	21
2.3 Data Types	
2.4 Data Arrangement	
2.5 Addressing Modes	
Table 2.5.1 Addressing Modes of M16C/80 Series 1	25
Table 2.5.2 Addressing Modes of M16C/80 Series 2	26
Table 2.5.3 Relative Address Ranges of Relative Addressing	34
2.6 Instruction Set	
Table 2.6.1 Generic format	47
Table 2.6.2 Quick format	47
Table 2.6.3 Short format	47
Table 2.6.4 Zero format	47
Table 2.6.5 4 Bit Transfer Instruction	68
Table 2.6.6 Conditional Store Instruction	69
Table 2.6.7 String Instruction	70
Table 2.6.8 Multiply Instruction	72
Table 2.6.9 Divide Instruction	73
Table 2.6.10 Difference between DIV and DIVX Instructions	74
Table 2.6.11 Decimal Add Instruction	75
Table 2.6.12 Decimal Subtract Instruction	76
Table 2.6.13 Sum of Products Calculate Instruction	77
Table 2.6.14 MAX, MIX, and CLIP instructions	78
Table 2.6.15 SCcnd instruction	79
Table 2.6.16 Unconditional Branch Instruction	80
Table 2.6.17 Indirect Branch Instruction	81
Table 2.6.18 Special Page Branch Instruction	82

Table 2.6.19 Conditional Branch Instruction	83
Table 2.6.20 Add (Subtract) & Conditional Branch Instruction	84
Table 2.6.21 Logical Bit Manipulating Instruction	85
Table 2.6.22 Conditional Bit Transfer Instruction	86
Table 2.6.23 Sign-extension instruction	87
Table 2.6.24 Index instruction	88
Table 2.6.25 Stack Frame Build Instruction	90
Table 2.6.26 Deallocate Stack Frame Instruction	91
Table 2.6.27 OS Support Instructions	92

2.7 Outline of Interrupt

Chapter 3 Functions of Assembler

3.1 Outline of AS308 System

Table 3.1.1 List of Input/output Files	108
--	-----

3.2 Method for Writing Source Program

Table 3.2.1 Types of Names Defined by User	112
Table 3.2.2 Description of Operands	113
Table 3.2.3 Description Range of Floating-point Numbers	114
Table 3.2.4 List of Operators	115
Table 3.2.5 Calculation Priority	115
Table 3.2.6 Types of Lines	116
Table 3.2.7 Types of Sections	118
Table 3.2.8 Section Attributes	119
Table 3.2.9 Replacement Instruction List	140

Chapter 4 Programming Style

4.1 Hardware Definition

4.2 Initial Setting the CPU

4.3 Setting when using Interrupts

4.4 Dividing Source File

4.5 A Little Tips...(Programing technique)

4.6 Standard processing program

Appendix Command input form and command parameters in AS308 system

Appendix A Generating Object Files

Table A.1	Command Options of as308	Appendix-4
Table A.2	Command Options of ln308	Appendix-9
Table A.3	Command Options of lmc308	Appendix-12

Chapter 1

Overview of M16C/80 Series

- 1.1 Features of M16C/80 Series
- 1.2 Outline of M16C/80 Group

1.1 Features of M16C/80 Series

The M16C/80 series is a line of single-chip microcomputers that have been developed for use in built-in equipment. This section describes the features of the M16C/80 series.

Features of the M16C/80 series

The M16C/80 series has its frequently used instructions placed in a 1-byte op-code. For this reason, it allows you to write a highly memory efficient program.

Furthermore, although the M16C/80 series is a 16-bit microcomputer, it can perform 1, 4, and 8-bit processing efficiently. Especially, 32-bit processing is handled more efficiently than in the M16C/60 series. The M16C/80 series has many instructions that can be executed in one clock period. For this reason, it is possible to write a high-speed processing program.

The M16C/80 series provides 1 M bytes of linear addressing space. Therefore, the M16C/80 series is also suitable for applications that require a large program size.

The features of the M16C/80 series can be summarized as follows:

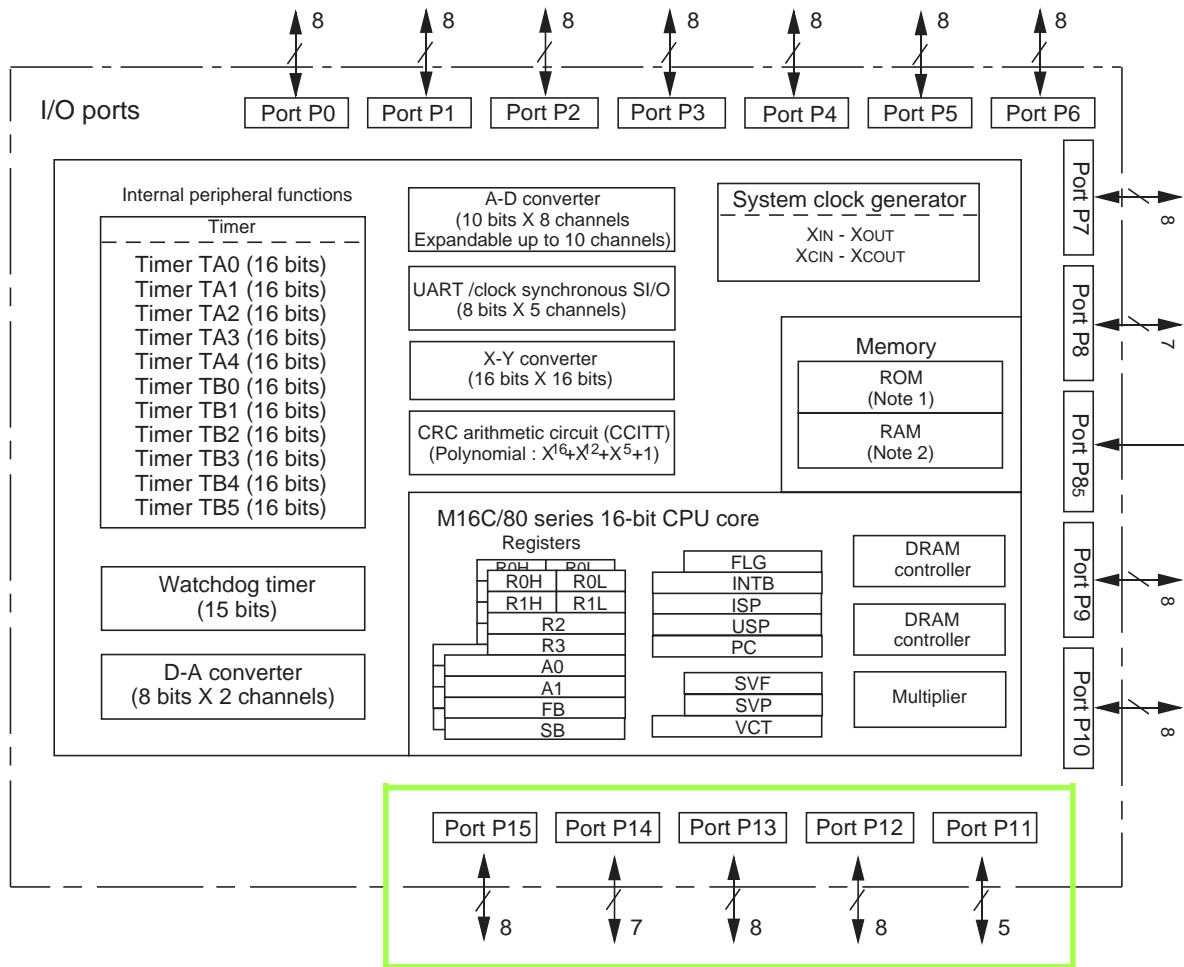
- (1) The M16C/80 series allows you to create a memory-efficient program without requiring a large memory capacity.
- (2) The M16C/80 series allows you to create a high-speed processing program.
- (3) The M16C/80 series provides 1 M bytes of addressing space, making it suitable for even large-capacity applications.

1.2 Outline of M16C/80 Group

This section introduces the M16C/80 group by way of explaining the internal configuration of the M16C/80 series. The M16C/80 group is a product that comprises the basis of the M16C/80 series. For details about this product, refer to the data sheets and user's manuals.

Internal Block Diagram

Figure 1.2.1 shows a block diagram of the M16C/80 group.



Note 1: ROM size depends on MCU type.

Note 2: RAM size depends on MCU type.

Note 3: Described is 144 pin version only.

Figure 1.2.1 Block diagram of the M16C/80 group

Outline Specifications of the M16C/80 Group

Table 1.2.1 lists the outline specifications of the M16C/80 group.

Table 1.2.1 Outline Specifications of M16C/80 Group

Item		Performance
Number of basic instructions		106 instructions
Shortest instruction execution time		50ns(f(XIN)=20MHz)
Memory capacity	ROM	128K bytes
	RAM	10K bytes
I/O port	P0 to P10 (except P85)	8 bits x 10, 7 bits x 1 (100-pin version)
	P0 to P15 (except P85)	8 bits x 10, 7 bits x 1, 5 bits x 1 (144-pin version)
Input port	P85	1 bit x 1
Multifunction timer	TA0, TA1, TA2, TA3, TA4	16 bits x 5
	TB0, TB1, TB2, TB3, TB4, TB5	16 bits x 6
Serial I/O	UART0, UART1, UART2, UART3, UART4	(UART or clock synchronous) x 5
A-D converter		10 bits x (8 + 2) channels
D-A converter		8 bits x 2
DMAC		4 channels
DRAM controller		CAS before RAS refresh, self-refresh, EDO, FP
CRC calculation circuit		CRC-CCITT
X-Y converter		16 bits X 16 bits
Watchdog timer		15 bits x 1 (with prescaler)
Interrupt		29 internal and 8 external sources, 4 software sources, 7 levels
Clock generating circuit		2 built-in clock generation circuits (built-in feedback resistor, and external ceramic or quartz oscillator)
Memory expansion		Available (up to 16 Mbytes)

Note: The above specifications are for the M30800MC. For details the memory size, refer to the data sheet and user's manual.

Register Structure

Table 1.2.2 shows the register structure of the M16C/80 series. Eight registers--R0, R1, R2, R3, A0, A1, SB, and FB--are available in two sets each. These sets are switched over by a register bank select flag.

Table 1.2.2 Register Structure of M16C/80 Series

Item	Content		
Register structure			
Data registers	<div>R0</div> <div>R1</div> <div>R2</div> <div>R3</div> <div><div>b15</div><div>b0</div></div>	<div>R2R0</div> <div>R3R1</div> <div><div>b31</div><div>b0</div></div>	<div>R0</div> <div>R1</div> <div><div>b7</div><div>b0</div><div>b7</div><div>b0</div></div>
Address registers	<div>A0</div> <div>A1</div> <div><div>b23</div><div>b0</div></div>		
Base registers	<div>SB</div> <div>FB</div> <div><div>b23</div><div>b0</div></div>		
Control registers	<div>PC</div> <div>INTB</div> <div>USP</div> <div>ISP</div> <div>FLG</div> <div><div>b23</div><div>b0</div><div>b15</div><div>b0</div></div>	<div>(Details of FLG)</div> <div><div>b15</div><div>IPL</div><div>b0</div></div> <div><div>U</div><div>I</div><div>O</div><div>B</div><div>S</div><div>Z</div><div>D</div><div>C</div></div> <div>IPL : Processor interrupt priority level (Levels 0 to 7; larger the number, higher the priority)</div> <div>U : Stack pointer select flag (ISP when U = 0, USP when U = 1)</div> <div>I : Interrupt enable flag (Enabled when I = 1)</div> <div>O : Overflow flag (0 = 1 when overflow occurs)</div> <div>B : Register bank select flag (Register bank 0 when B = 0, register bank 1 when B = 1)</div> <div>S : Sign flag (S = 1 when operation resulted in negative, S = 0 when positive)</div> <div>Z : Zero flag (Z = 1 when operation resulted in zero)</div> <div>D : Debug flag (Program is single-stepped when D = 1)</div> <div>C : Carry flag (carry or borrow)</div> <div><div> </div> : Reserved area</div>	
High-speed interrupt registers	<div>SVF</div> <div>SVP</div> <div>VCT</div> <div><div>b15</div><div>b0</div><div>b23</div><div>b0</div></div>		
DMAC related registers	<div>DMD0</div> <div>DMD1</div> <div>DCT0</div> <div>DCT1</div> <div>DRC0</div> <div>DRC1</div> <div>DMA0</div> <div>DMA1</div> <div>DSA0</div> <div>DSA1</div> <div>DRA0</div> <div>DRA1</div> <div><div>b7</div><div>b0</div><div>b15</div><div>b0</div><div>b23</div><div>b0</div></div>		

MEMO

Chapter 2

CPU Programming Model

- 2.1 Address Space
- 2.2 Register Sets
- 2.3 Data Types
- 2.4 Data Arrangement
- 2.5 Addressing Modes
- 2.6 Instruction Set
- 2.7 Outline of Interrupt

2.1 Address Space

The M16C/80 series has 16 M bytes of address space ranging from address 000000H to address FFFFFFFH. This section explains the address space and memory mapping, the SFR area, and the fixed vector area of the M16C/80 group.

Address Space

Figure 2.1.1 shows the address space of the M16C/80 group.

Addresses 000000H to 0003FFH are the Special Function Register (SFR) area. The SFR area in each type of M16C/80 group microcomputer begins with address 0003FFH and expands toward smaller addresses.

Addresses following 000400H constitute the memory area. The memory area in each type of M16C/80 group microcomputer consists of a RAM area which begins with address 00400H and expands toward larger addresses and a ROM area which begins with address FFFFFFFH and expands toward smaller addresses. However, addresses FFEE00H to FFFFFFFH are the fixed vector area.

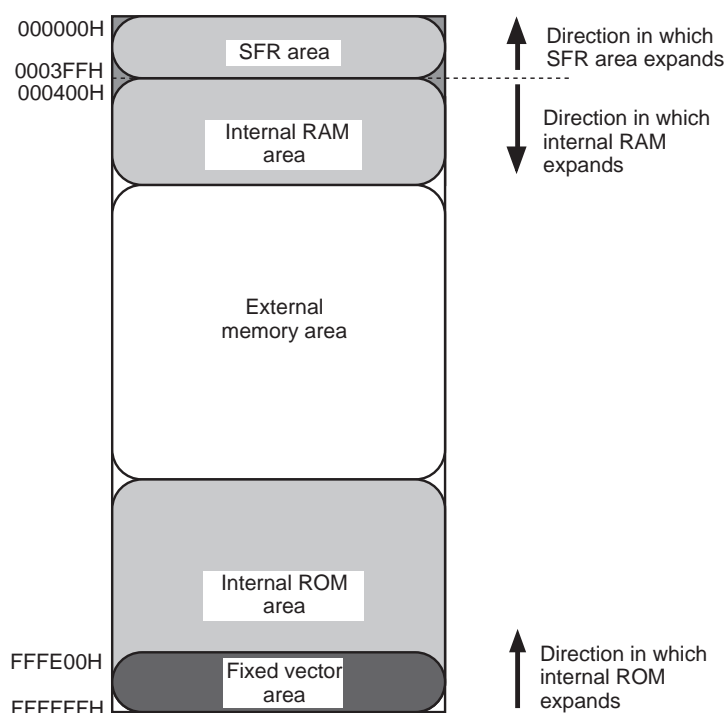


Figure 2.1.1 Address space

2.1.1 Operation Modes and Memory Mapping

The M16C/80 group chooses one operation mode from three modes available: single-chip, memory expansion, and microprocessor modes. The M16C/80 group address space and the usable areas and memory mapping varies with each operation mode.

Operation Modes and Memory Mapping

- Single-chip mode
In this mode, only the internal areas (SFR, internal RAM, and internal ROM) can be accessed.
- Memory expansion mode
In this mode, the internal areas (SFR, internal RAM, and internal ROM) and an external memory area can be accessed.
- Microprocessor mode
In this mode, the SFR and internal RAM areas and an external memory area can be accessed.
(The internal ROM area cannot be accessed.)

Figure 2.1.2 shows the M16C/80 group memory mapping in each operation mode.

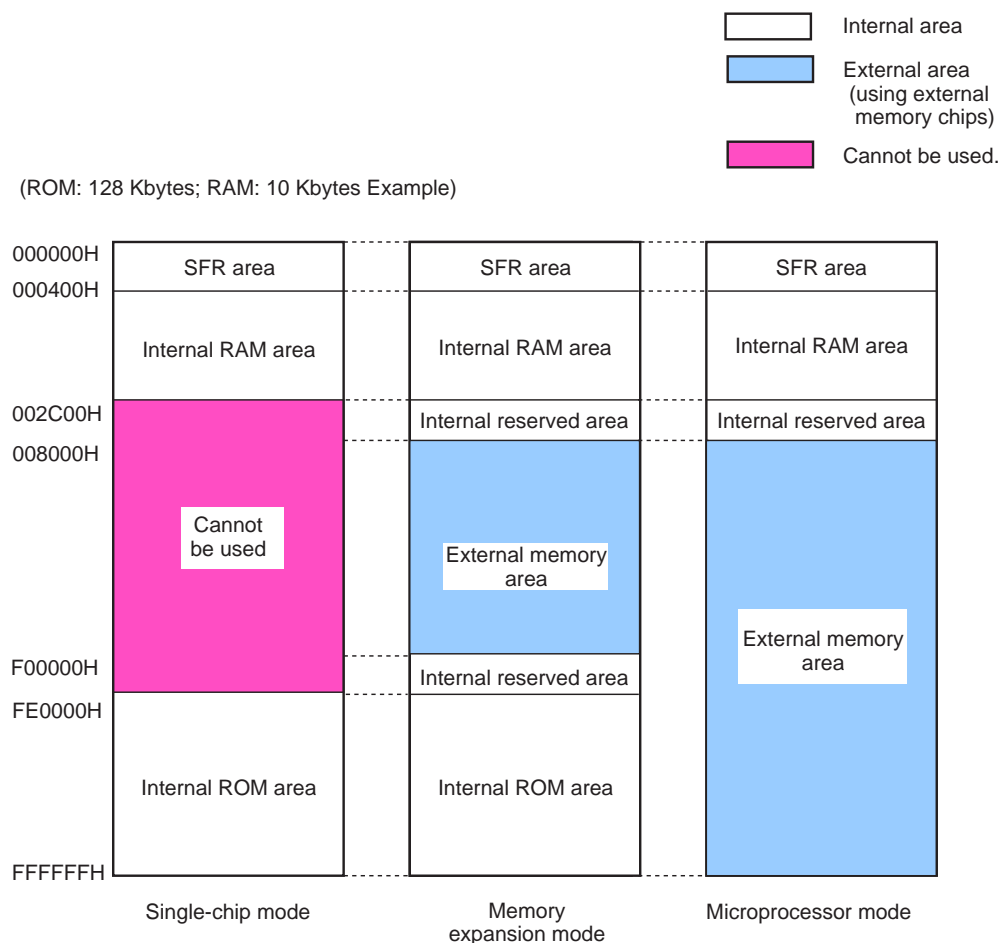


Figure 2.1.2 Operation modes and memory mapping

2.1.2 SFR Area

A range of control registers are allocated in this area, including the processor mode register that determines the operation mode and the peripheral unit control registers for I/O ports, A-D converter, UART, and timers. For the bit configurations of these control registers, refer to the M16C/80 group data sheets and user's manuals.

The unused locations in the SFR area are reserved for the system and cannot be used by the user.

SFR Area: Control Register Allocation (100-pin version)

Figures 2.1.3, 2.1.4, 2.1.5 and 2.1.6 show control register allocations in the SFR area.

0000 ₁₆		0060 ₁₆	
0001 ₁₆		0061 ₁₆	
0002 ₁₆		0062 ₁₆	
0003 ₁₆		0063 ₁₆	
0004 ₁₆	Processor mode register 0 (PM0)	0064 ₁₆	
0005 ₁₆	Processor mode register 1 (PM1)	0065 ₁₆	
0006 ₁₆	System clock control register 0 (CM0)	0066 ₁₆	
0007 ₁₆	System clock control register 1 (CM1)	0067 ₁₆	
0008 ₁₆	Wait control register (WCR)	0068 ₁₆	DMA0 interrupt control register (DM0IC)
0009 ₁₆	Address match interrupt enable register (AIER)	0069 ₁₆	Timer B5 interrupt control register (TB5IC)
000A ₁₆	Protect register (PRCR)	006A ₁₆	DMA2 interrupt control register (DM1IC)
000B ₁₆	External data bus width control register (DS)	006B ₁₆	UART2 receive/ACK interrupt control register (S2RIC)
000C ₁₆	Main clock division register (MCD)	006C ₁₆	Timer A0 interrupt control register (TA0IC)
000D ₁₆		006D ₁₆	UART3 receive/ACK interrupt control register (S3RIC)
000E ₁₆	Watchdog timer start register (WDTS)	006E ₁₆	Timer A2 interrupt control register (TA2IC)
000F ₁₆	Watchdog timer control register (WDC)	006F ₁₆	UART4 receive/ACK interrupt control register (S4RIC)
0010 ₁₆		0070 ₁₆	Timer A4 interrupt control register (TA4IC)
0011 ₁₆	Address match interrupt register 0 (RMAD0)	0071 ₁₆	Bus collision detection(UART3) interrupt control register (BCN3IC)
0012 ₁₆		0072 ₁₆	UART0 receive interrupt control register (S0RIC)
0013 ₁₆		0073 ₁₆	A-D conversion interrupt control register (ADIC)
0014 ₁₆		0074 ₁₆	UART1 receive interrupt control register (S1RIC)
0015 ₁₆	Address match interrupt register 1 (RMAD1)	0075 ₁₆	
0016 ₁₆		0076 ₁₆	Timer B1 interrupt control register (TB1IC)
0017 ₁₆		0077 ₁₆	
0018 ₁₆		0078 ₁₆	Timer B3 interrupt control register (TB3IC)
0019 ₁₆	Address match interrupt register 2 (RMAD2)	0079 ₁₆	
001A ₁₆		007A ₁₆	INT5 interrupt control register (INT5IC)
001B ₁₆		007B ₁₆	
001C ₁₆		007C ₁₆	INT3 interrupt control register (INT3IC)
001D ₁₆	Address match interrupt register 3 (RMAD3)	007D ₁₆	
001E ₁₆		007E ₁₆	INT1 interrupt control register (INT1IC)
001F ₁₆		007F ₁₆	
0020 ₁₆		0080 ₁₆	
0021 ₁₆	Emulator interrupt vector table register (EIAD) *	0081 ₁₆	
0022 ₁₆		0082 ₁₆	
0023 ₁₆	Emulator interrupt detect register (EITD) *	0083 ₁₆	
0024 ₁₆	Emulator protect register (EPRR) *	0084 ₁₆	
0025 ₁₆		0085 ₁₆	
0026 ₁₆		0086 ₁₆	
0027 ₁₆		0087 ₁₆	
0028 ₁₆		0088 ₁₆	DMA1 interrupt control register (DM1IC)
0029 ₁₆		0089 ₁₆	UART2 transmit/NACK interrupt control register (S2TIC)
002A ₁₆		008A ₁₆	DMA3 interrupt control register (DM3IC)
002B ₁₆		008B ₁₆	UART3 transmit/NACK interrupt control register (S3TIC)
002C ₁₆		008C ₁₆	Timer A1 interrupt control register (TA1IC)
002D ₁₆		008D ₁₆	UART4 transmit/NACK interrupt control register (S4TIC)
002E ₁₆		008E ₁₆	Timer A3 interrupt control register (TA3IC)
002F ₁₆		008F ₁₆	Bus collision detection(UART2) interrupt control register (BCN2IC)
0030 ₁₆	ROM areaset register (ROA) *	0090 ₁₆	UART0 transmit interrupt control register (S0TIC)
0031 ₁₆	Debug monitor area set register (DBA) *	0091 ₁₆	Bus collision detection(UART4) interrupt control register (BCN4IC)
0032 ₁₆	Expansion area set register 0 (EXA0) *	0092 ₁₆	UART1 transmit interrupt control register (S1TIC)
0033 ₁₆	Expansion area set register 1 (EXA1) *	0093 ₁₆	Key input interrupt control register (KUPIC)
0034 ₁₆	Expansion area set register 2 (EXA2) *	0094 ₁₆	Timer B0 interrupt control register (TB0IC)
0035 ₁₆	Expansion area set register 3 (EXA3) *	0095 ₁₆	
0036 ₁₆		0096 ₁₆	Timer B2 interrupt control register (TB2IC)
0037 ₁₆		0097 ₁₆	
0038 ₁₆		0098 ₁₆	Timer B4 interrupt control register (TB4IC)
0039 ₁₆		0099 ₁₆	
003A ₁₆		009A ₁₆	INT4 interrupt control register (INT4IC)
003B ₁₆		009B ₁₆	
003C ₁₆		009C ₁₆	INT2 interrupt control register (INT2IC)
003D ₁₆		009D ₁₆	
003E ₁₆		009E ₁₆	INT0 interrupt control register (INT0IC)
003F ₁₆		009F ₁₆	Exit priority register (RLVL)
0040 ₁₆	DRAM control register (DRAMCONT)	00A0 ₁₆	
0041 ₁₆	DRAM refresh interval set register (REFCNT)	00A1 ₁₆	
0042 ₁₆		00A2 ₁₆	
0043 ₁₆		00A3 ₁₆	
0044 ₁₆		00A4 ₁₆	

* As this register is used exclusively for debugger purposes, user cannot use this. Do not access to the register.

Figure 2.1.3 Control register allocation 1

02C0 ₁₆	X0 register (X0R) Y0 register (Y0R)	0300 ₁₆	Timer B3, 4, 5 count start flag (TBSR)
02C1 ₁₆		0301 ₁₆	
02C2 ₁₆	X1 register (X1R) Y1 register (Y1R)	0302 ₁₆	
02C3 ₁₆		0303 ₁₆	Timer A1-1 register (TA11)
02C4 ₁₆	X2 register (X2R) Y2 register (Y2R)	0304 ₁₆	
02C5 ₁₆		0305 ₁₆	Timer A2-1 register (TA21)
02C6 ₁₆	X3 register (X3R) Y3 register (Y3R)	0306 ₁₆	
02C7 ₁₆		0307 ₁₆	Timer A4-1 register (TA41)
02C8 ₁₆	X4 register (X4R) Y4 register (Y4R)	0308 ₁₆	Three-phase PWM control register 0(INVC0)
02C9 ₁₆		0309 ₁₆	Three-phase PWM control register 1(INVC1)
02CA ₁₆	X5 register (X5R) Y5 register (Y5R)	030A ₁₆	Three-phase output buffer register 0(IDB0)
02CB ₁₆		030B ₁₆	Three-phase output buffer register 1(IDB1)
02CC ₁₆	X6 register (X6R) Y6 register (Y6R)	030C ₁₆	Dead time timer(DTT)
02CD ₁₆		030D ₁₆	Timer B2 interrupt occurrence frequency set counter(ICTB2)
02CE ₁₆	X7 register (X7R) Y7 register (Y7R)	030E ₁₆	
02CF ₁₆		030F ₁₆	
02D0 ₁₆	X8 register (X8R) Y8 register (Y8R)	0310 ₁₆	
02D1 ₁₆		0311 ₁₆	Timer B3 register (TB3)
02D2 ₁₆	X9 register (X9R) Y9 register (Y9R)	0312 ₁₆	
02D3 ₁₆		0313 ₁₆	Timer B4 register (TB4)
02D4 ₁₆	X10 register (X10R) Y10 register (Y10R)	0314 ₁₆	
02D5 ₁₆		0315 ₁₆	Timer B5 register (TB5)
02D6 ₁₆	X11 register (X11R) Y11 register (Y11R)	0316 ₁₆	
02D7 ₁₆		0317 ₁₆	
02D8 ₁₆	X12 register (X12R) Y12 register (Y12R)	0318 ₁₆	
02D9 ₁₆		0319 ₁₆	
02DA ₁₆	X13 register (X13R) Y13 register (Y13R)	031A ₁₆	
02DB ₁₆		031B ₁₆	Timer B3 mode register (TB3MR)
02DC ₁₆	X14 register (X14R) Y14 register (Y14R)	031C ₁₆	Timer B4 mode register (TB4MR)
02DD ₁₆		031D ₁₆	Timer B5 mode register (TB5MR)
02DE ₁₆	X15 register (X15R) Y15 register (Y15R)	031E ₁₆	
02DF ₁₆		031F ₁₆	Interrupt cause select register (IFSR)
02E0 ₁₆	XY control register (XYC)	0320 ₁₆	
02E1 ₁₆		0321 ₁₆	
02E2 ₁₆		0322 ₁₆	
02E3 ₁₆		0323 ₁₆	
02E4 ₁₆		0324 ₁₆	
02E5 ₁₆		0325 ₁₆	UART3 special mode register 3 (U3SMR3)
02E6 ₁₆		0326 ₁₆	UART3 special mode register 2 (U3SMR2)
02E7 ₁₆		0327 ₁₆	UART3 special mode register (U3SMR)
02E8 ₁₆		0328 ₁₆	UART3 transmit/receive mode register (U3MR)
02E9 ₁₆		0329 ₁₆	UART3 bit rate generator (U3BRG)
02EA ₁₆		032A ₁₆	UART3 transmit buffer register (U3TB)
02EB ₁₆		032B ₁₆	
02EC ₁₆		032C ₁₆	UART3 transmit/receive control register 0 (U3C0)
02ED ₁₆		032D ₁₆	UART3 transmit/receive control register 1 (U3C1)
02EE ₁₆		032E ₁₆	UART3 receive buffer register (U3RB)
02EF ₁₆		032F ₁₆	
02F0 ₁₆		0330 ₁₆	
02F1 ₁₆		0331 ₁₆	
02F2 ₁₆		0332 ₁₆	
02F3 ₁₆		0333 ₁₆	
02F4 ₁₆		0334 ₁₆	
02F5 ₁₆	UART4 special mode register 3 (U4SMR3)	0335 ₁₆	UART2 special mode register 3 (U2SMR3)
02F6 ₁₆	UART4 special mode register 2 (U4SMR2)	0336 ₁₆	UART2 special mode register 2 (U2SMR2)
02F7 ₁₆	UART4 special mode register (U4SMR)	0337 ₁₆	UART2 special mode register (U2SMR)
02F8 ₁₆	UART4 transmit/receive mode register (U4MR)	0338 ₁₆	UART2 transmit/receive mode register (U2MR)
02F9 ₁₆	UART4 bit rate generator (U4BRG)	0339 ₁₆	UART2 bit rate generator (U2BRG)
02FA ₁₆	UART4 transmit buffer register (U4TB)	033A ₁₆	UART2 transmit buffer register (U2TB)
02FB ₁₆		033B ₁₆	
02FC ₁₆	UART4 transmit/receive control register 0 (U4C0)	033C ₁₆	UART2 transmit/receive control register 0 (U2C0)
02FD ₁₆	UART4 transmit/receive control register 1 (U4C1)	033D ₁₆	UART2 transmit/receive control register 1 (U2C1)
02FE ₁₆		033E ₁₆	
02FF ₁₆	UART4 receive buffer register (U4RB)	033F ₁₆	UART2 receive buffer register (U2RB)

Figure 2.1.4 Control register allocation 2

0340 ₁₆	Count start flag (TABSR)	0380 ₁₆	A-D register 0 (AD0)
0341 ₁₆	Clock prescaler reset flag (CPSRF)	0381 ₁₆	
0342 ₁₆	One-shot start flag (ONSF)	0382 ₁₆	A-D register 1 (AD1)
0343 ₁₆	Trigger select register (TRGSR)	0383 ₁₆	
0344 ₁₆	Up-down flag (UDF)	0384 ₁₆	A-D register 2 (AD2)
0345 ₁₆		0385 ₁₆	
0346 ₁₆	Timer A0 (TA0)	0386 ₁₆	A-D register 3 (AD3)
0347 ₁₆		0387 ₁₆	
0348 ₁₆	Timer A1 (TA1)	0388 ₁₆	A-D register 4 (AD4)
0349 ₁₆		0389 ₁₆	
034A ₁₆	Timer A2 (TA2)	038A ₁₆	A-D register 5 (AD5)
034B ₁₆		038B ₁₆	
034C ₁₆	Timer A3 (TA3)	038C ₁₆	A-D register 6 (AD6)
034D ₁₆		038D ₁₆	
034E ₁₆	Timer A4 (TA4)	038E ₁₆	A-D register 7 (AD7)
034F ₁₆		038F ₁₆	
0350 ₁₆	Timer B0 (TB0)	0390 ₁₆	
0351 ₁₆		0391 ₁₆	
0352 ₁₆	Timer B1 (TB1)	0392 ₁₆	
0353 ₁₆		0393 ₁₆	
0354 ₁₆	Timer B2 (TB2)	0394 ₁₆	A-D control register 2 (ADCON2)
0355 ₁₆		0395 ₁₆	
0356 ₁₆	Timer A0 mode register (TA0MR)	0396 ₁₆	A-D control register 0 (ADCON0)
0357 ₁₆	Timer A1 mode register (TA1MR)	0397 ₁₆	A-D control register 1 (ADCON1)
0358 ₁₆	Timer A2 mode register (TA2MR)	0398 ₁₆	D-A register 0 (DA0)
0359 ₁₆	Timer A3 mode register (TA3MR)	0399 ₁₆	
035A ₁₆	Timer A4 mode register (TA4MR)	039A ₁₆	D-A register 1 (DA1)
035B ₁₆	Timer B0 mode register (TB0MR)	039B ₁₆	
035C ₁₆	Timer B1 mode register (TB1MR)	039C ₁₆	D-A control register (DACON)
035D ₁₆	Timer B2 mode register (TB2MR)	039D ₁₆	
035E ₁₆		039E ₁₆	
035F ₁₆		039F ₁₆	
0360 ₁₆	UART0 transmit/receive mode register (U0MR)	03A0 ₁₆	
0361 ₁₆	UART0 bit rate generator (U0BRG)	03A1 ₁₆	
0362 ₁₆	UART0 transmit buffer register (U0TB)	03A2 ₁₆	
0363 ₁₆		03A3 ₁₆	
0364 ₁₆	UART0 transmit/receive control register 0 (U0C0)	03A4 ₁₆	
0365 ₁₆	UART0 transmit/receive control register 1 (U0C1)	03A5 ₁₆	
0366 ₁₆		03A6 ₁₆	
0367 ₁₆	UART0 receive buffer register (U0RB)	03A7 ₁₆	
0368 ₁₆	UART1 transmit/receive mode register (U1MR)	03A8 ₁₆	
0369 ₁₆	UART1 bit rate generator (U1BRG)	03A9 ₁₆	
036A ₁₆		03AA ₁₆	
036B ₁₆	UART1 transmit buffer register (U1TB)	03AB ₁₆	
036C ₁₆	UART1 transmit/receive control register 0 (U1C0)	03AC ₁₆	
036D ₁₆	UART1 transmit/receive control register 1 (U1C1)	03AD ₁₆	
036E ₁₆		03AE ₁₆	
036F ₁₆	UART1 receive buffer register (U1RB)	03AF ₁₆	Function select register C (PSC)
0370 ₁₆	UART transmit/receive control register 2 (UCON2)	03B0 ₁₆	Function select register A0 (PS0)
0371 ₁₆		03B1 ₁₆	Function select register A1 (PS1)
0372 ₁₆		03B2 ₁₆	Function select register B0 (PSL0)
0373 ₁₆		03B3 ₁₆	Function select register B1 (PSL1)
0374 ₁₆		03B4 ₁₆	Function select register A2 (PS2)
0375 ₁₆		03B5 ₁₆	Function select register A3 (PS3)
0376 ₁₆	Flash memory control register 1 (FMR1) (Note)	03B6 ₁₆	Function select register B2 (PSL2)
0377 ₁₆	Flash memory control register 0 (FMR0) (Note)	03B7 ₁₆	Function select register B3 (PSL3)
0378 ₁₆	DMA0 request cause select register (DM0SL)	03B8 ₁₆	
0379 ₁₆	DMA1 request cause select register (DM1SL)	03B9 ₁₆	
037A ₁₆	DMA2 request cause select register (DM2SL)	03BA ₁₆	
037B ₁₆	DMA3 request cause select register (DM3SL)	03BB ₁₆	
037C ₁₆		03BC ₁₆	
037D ₁₆	CRC data register (CRCD)	03BD ₁₆	
037E ₁₆	CRC input register (CRCIN)	03BE ₁₆	
037F ₁₆		03BF ₁₆	

Note :This register exists in the flash memory version.

Figure 2.1.5 Control register allocation 3

03C0H	Port P6(P6)
03C1H	Port P7(P7)
03C2H	Port P6 direction register (PD6)
03C3H	Port P7 direction register (PD7)
03C4H	Port P8(P8)
03C5H	Port P9(P9)
03C6H	Port P8 direction register (PD8)
03C7H	Port P9 direction register (PD9)
03C8H	Port P10(P10)
03C9H	
03CAH	Port P10 direction register (PD10)
03CBH	
03CCH	
03CDH	
03CEH	
03CFH	
03D0H	
03D1H	
03D2H	
03D3H	
03D4H	
03D5H	
03D6H	
03D7H	
03D8H	
03D9H	
03DAH	Pull-up control register 2(PUR2)
03DBH	Pull-up control register 3(PUR3)
03DCH	
03DDH	
03DEH	
03DFH	
03E0H	Port P0(P0)
03E1H	Port P1(P1)
03E2H	Port P0 direction register (PD0)
03E3H	Port P1 direction register (PD1)
03E4H	Port P2(P2)
03E5H	Port P3(P3)
03E6H	Port P2 direction register (PD2)
03E7H	Port P3 direction register (PD3)
03E8H	Port P4(P4)
03E9H	Port P5(P5)
03EAH	Port P4 direction register (PD4)
03EBH	Port P5 direction register (PD5)
03ECH	
03EDH	
03EEH	
03EFH	
03F0H	Pull-up control register 0(PUR0)
03F1H	Pull-up control register 1(PUR1)
03F2H	
03F3H	
03F4H	
03F5H	
03F6H	
03F7H	
03F8H	
03F9H	
03FAH	
03FBH	
03FCH	
03FDH	
03FEH	
03FFH	Port control register (PCR)

Note: Address 03C9H, 03CBH to 03D3H area is for future plan.
Must set "FF₁₆" to address 03CBH, 03CEH, 03CFH, 03D2H, 03D3H at initial setting.

Figure 2.1.6 Control register allocation 4

Determination of Operation Mode

The operation modes of the M16C/80 group are determined by the CNVSS pin, processor mode register 0 (address 000004H), and bits 0 and 1.

Figure 2.1.7 shows the configuration of processor mode register 0.

Processor mode register 0 (Note 1)

b7	b6	b5	b4	b3	b2	b1	b0	Symbol	Address	When reset		
	0							PM0	0004 ₁₆	80 ₁₆		
								Bit symbol	Bit name	Function	R	W
								PM00	Processor mode bit	b1 b0 0 0 : Single-chip mode 0 1 : Memory expansion mode 1 0 : Inhibited 1 1 : Microprocessor mode	<input type="radio"/>	<input type="radio"/>
								PM01			<input type="radio"/>	<input type="radio"/>
								PM02	R/W mode select bit(Note 7)	0 : <u>RD</u> , <u>BHE</u> , <u>WR</u> 1 : RD , WRH , WRL	<input type="radio"/>	<input type="radio"/>
								PM03	Software reset bit	The device is reset when this bit is set to "1". The value of this bit is "0" when read.	<input type="radio"/>	<input type="radio"/>
								PM04	Multiplexed bus space select bit (Note 3)	b4 b5 0 0 : Multiplexed bus is not used 0 1 : Allocated to <u>CS2</u> space 1 0 : Allocated to <u>CS1</u> space 1 1 : Allocated to entire space (Note 4)	<input type="radio"/>	<input type="radio"/>
								PM05			<input type="radio"/>	<input type="radio"/>
								Reserved bit		Must always be set to "0".	<input type="radio"/>	<input type="radio"/>
								PM07	BCLK output disable bit (Note 5)	0 : BCLK is output (Note 6) 1 : Function set by bit 0,1 of system clock control register 0	<input type="radio"/>	<input type="radio"/>

Note 1 : Set bit 1 of protect register (address 000A₁₆) to "1" when writing new value to this register.

Note 2 : If the Vcc voltage is applied to the CNVss, the value of this register when reset when reset is 03₁₆.
(PM00 is set to "1" and PM07 is set to "0".)

Note 3 : Valid in microprocessor and memory expansion modes 1,2 and 3. Do not use multiplex bus when mode 0 is selected. Do not set to allocated to $\overline{CS2}$ space when mode 2 is selected.

Note 4 : After the reset has been released the M16C/80 group MCU operates using the separate bus. As a result, in microprocessor mode, you cannot select the full \overline{CS} space multiplex bus. When you select the full \overline{CS} space multiplex bus in memory expansion mode, the address bus operates with 64 K bytes boundance for each chip select.

Mode 0 : Multiplexed bus cannot be used.

Mode 1 : $\overline{CS0}$ to $\overline{CS2}$ when you select full \overline{CS} space.

Mode 2 : $\overline{CS0}$ to $\overline{CS1}$ when you select full \overline{CS} space.

Mode 3 : $\overline{CS0}$ to $\overline{CS3}$ when you select full \overline{CS} space.

Note 5 : No BCLK is output in single chip mode even when "0" is set in PM07. When stopping clock output in microprocessor or memory expansion mode, make the following settings: PM07="1", bit 0 (CM00) and bit 1 (CM01) of system clock control register 0 = "0". "L" is now output from P5₃.

Note 6 : See the data sheet for BCLK.

Note 7 : When using 16-bit bus width in DRAM controller, set this bit to "1".

Figure 2.1.7 Processor mode register 0

2.1.3 Fixed Vector Area

The M16C/80 group fixed vector area consists of addresses FFFE00H to FFFFFFFH.

Addresses FFFE00H to FFFFDDBH in this area constitute a special page vector table. This table is used to store the start addresses of subroutines and jump addresses, so that subroutine call and jump instructions can be executed using two bytes, helping to reduce the number of program steps.

Addresses FFFFDDBH to FFFFFFFH in the fixed vector area constitute a fixed interrupt vector table for reset and NMI. This table is used to store the start addresses of interrupt routines. An interrupt vector table for timer interrupts, etc. can be set at any desired address by an internal register (INTB). For details, refer to the section dealing with interrupts in Chapter 4.

Memory Mapping in Fixed Vector Area

Figure 2.1.8 shows memory mapping for the special page vector table and fixed vector area.

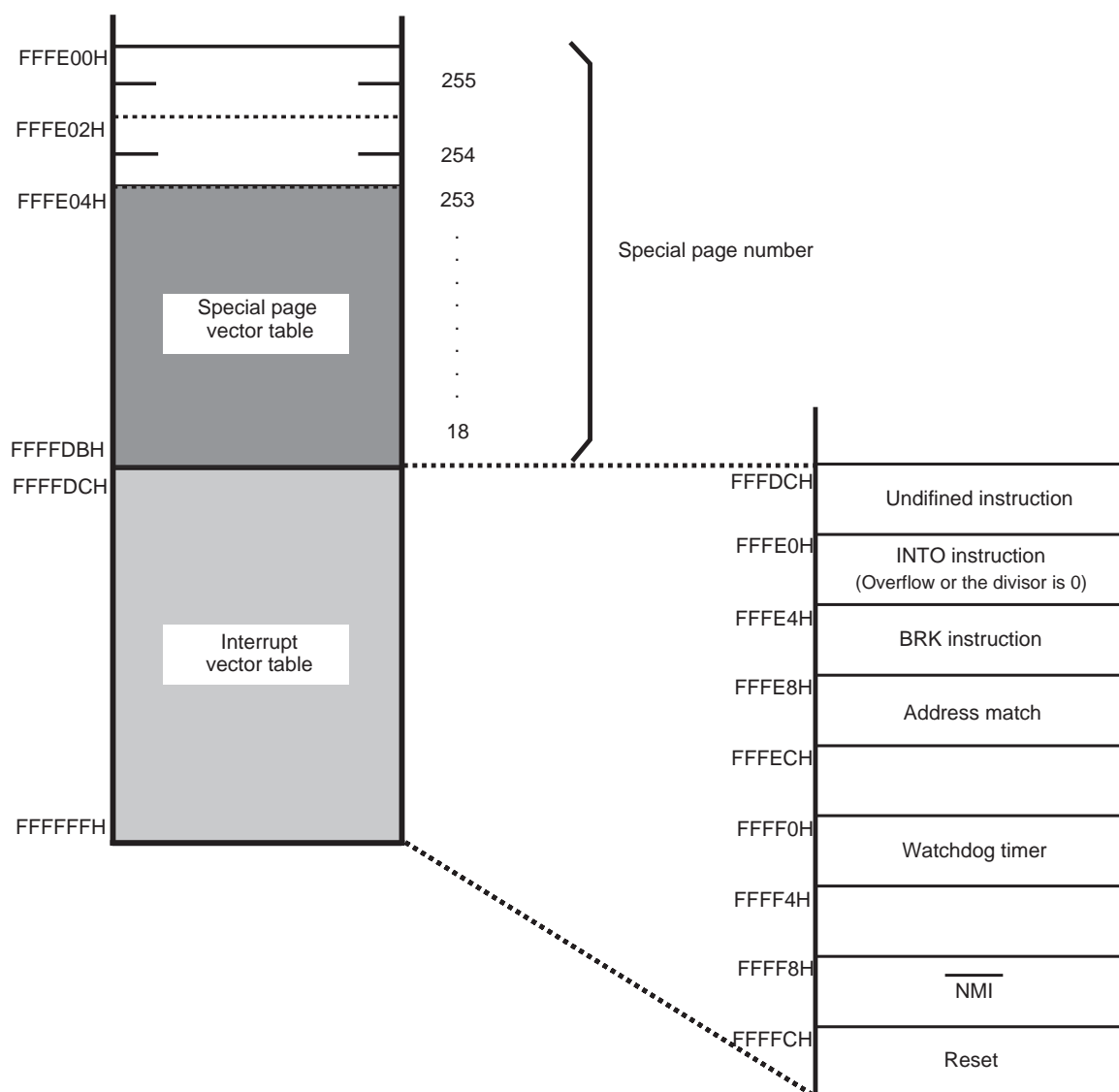


Figure 2.1.8 Memory mapping in fixed vector area

2.2 Register Set

The following explains the general registers, high-speed interrupt registers, and DMAC-related registers included in the M16C/80 series CPU core.

Register Structure

Figure 2.2.1 shows the register structure of the M16C/80 series CPU core. Eight registers--R0, R1, R2, R3, A0, A1, FB, and SB--are available in two sets each. The following shows the function of each register.

General registers

(1) Data registers (R0, R1, R2, and R3)

These registers consist of 16 bits each and are used mainly for data transfer and arithmetic/logic operations.

Registers R0 and R1 can be used separately for upper bytes (R0H, R1H) and lower bytes (R0L, R1L) as 8-bit data registers. For some instructions, registers R2 and R0 and registers R3 and R1 can be combined for use as 32-bit data registers (R2R0, R3R1), respectively.

(2) Address registers (A0 and A1)

These registers consist of 24 bits, and have the functions equivalent to those of the data registers. In addition, these registers are used in address register indirect addressing and address register relative addressing.

(3) Frame base register (FB)

This register consists of 24 bits, and is used in FB relative addressing.

(4) Static base register (SB)

This register consists of 24 bits, and is used in SB relative addressing.

(5) Program counter (PC)

This counter consists of 24 bits, indicating the address of an instruction to be executed.

(6) Interrupt table register (INTB)

This register consists of 24 bits, indicating the start address of an interrupt vector table.

(7) Stack pointers (USP or ISP)

There are two stack pointers: a user stack pointer (USP) and an interrupt stack pointer (ISP). Both of these pointers consist of 24 bits.

The stack pointers used (USP or ISP) are switched over by a stack pointer select flag (U flag). The U flag is assigned to bit 7 of the flag register (FLG).

Set odd numbers in USP and ISP. Execution efficiency is better when odd numbers are set.

(8) Flag register (FLG)

This register consists of 11 bits, each of which is used as a flag.

High-speed interrupt registers**(9) Save flag register (SVF)**

This register consists of 16 bits and is used to save the flag register when a high-speed interrupt is generated.

(10) Save PC register (SVP)

This register consists of 24 bits and is used to save the program counter when a high-speed interrupt is generated.

(11) Vector register (VCT)

This register consists of 24 bits and is used to indicate the jump address when a high-speed interrupt is generated.

DMAC related registers**(12) DMA mode registers (DMD0 and DMD1)**

These registers consist of 8 bits and are used to set the transfer mode, etc. for DMA.

(13) DMA transfer count registers (DCT0 and DCT1)

These registers consist of 16 bits and are used to set the number of DMA transfers performed.

(14) DMA transfer count reload registers (DRC0 and DRC1)

These registers consist of 16 bits and are used to reload the DMA transfer count registers.

(15) DMA memory address registers (DMA0 and DMA1)

These registers consist of 24 bits and are used to set a memory address at the source or destination of DMA transfer.

(16) DMA SFR address registers (DSA0 and DSA1)

These registers consist of 24 bits and are used to set a fixed address at the source or destination of DMA transfer.

(17) DMA memory address reload registers (DRA0 and DRA1)

These registers consist of 24 bits and are used to reload the DMA memory address registers.

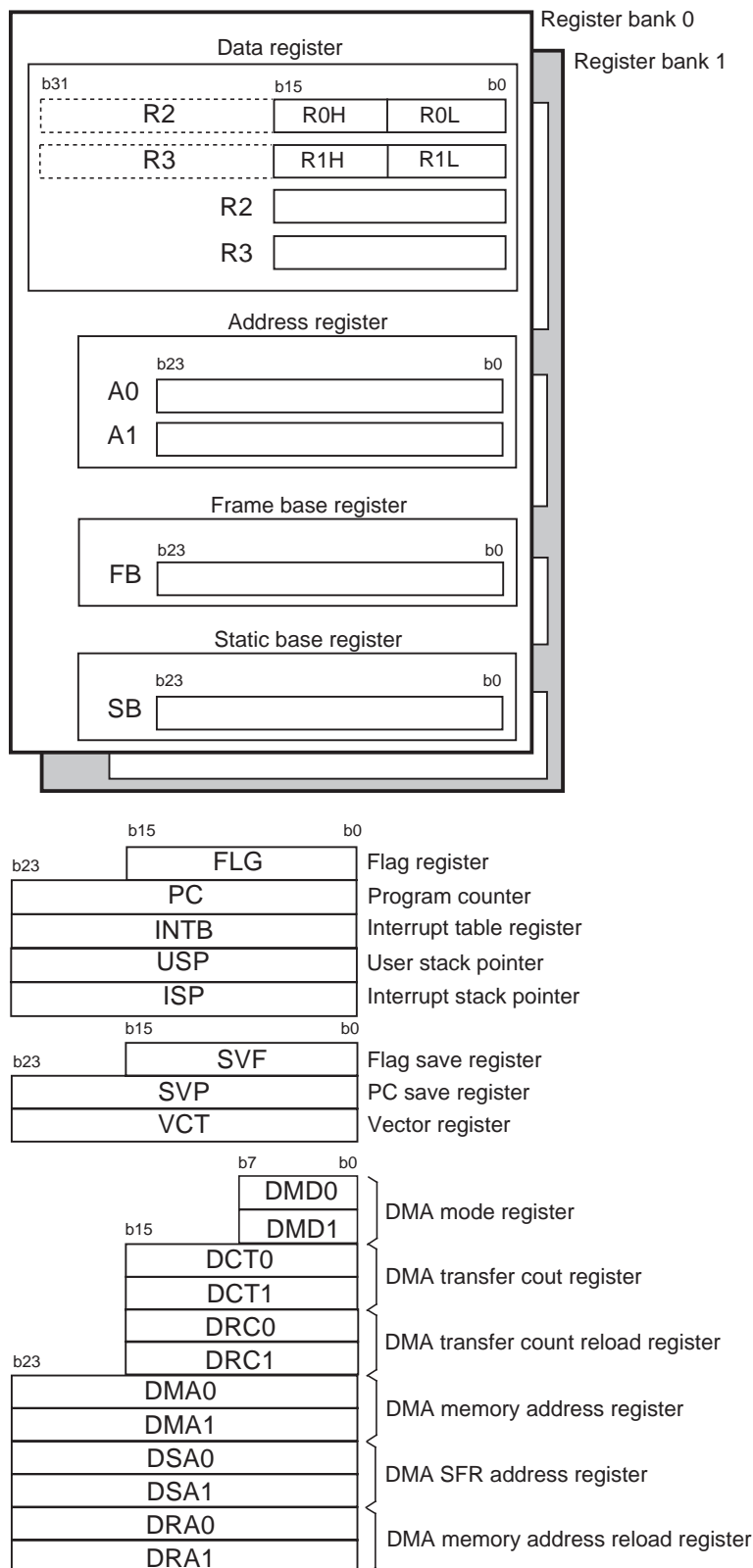


Figure 2.2.1 Register structure

Flag Register (FLG)

Figure 2.2.2 shows the bit configuration of the flag register (FLG). The function of each flag is described below.

- **Bit 0: Carry flag (C flag)**

This bit holds a carry or borrow that has occurred in an arithmetic/logic operation or a bit that has been shifted out.

- **Bit 1: Debug flag (D flag)**

This flag enables a single-step interrupt.

When this flag is 1, a single-step interrupt is generated after instruction execution. When the interrupt is accepted, this flag is cleared to 0.

- **Bit 2: Zero flag (Z flag)**

This flag is set to 1 when the operation resulted in 0; otherwise, the flag is 0.

- **Bit 3: Sign flag (S flag)**

This flag is set to 1 when the operation resulted in a negative number. The flag is 0 when the result is positive.

- **Bit 4: Register bank specifying flag (B flag)**

This flag chooses a register bank. Register bank 0 is selected when the flag is 0. Register bank 1 is selected when the flag is 1.

- **Bit 5: Overflow flag (O flag)**

This flag is set to 1 when the operation resulted in an overflow.

- **Bit 6: Interrupt enable flag (I flag)**

This flag enables a maskable interrupt.

The interrupt is enabled when the flag is 1, and is disabled when the flag is 0. This flag is cleared to 0 when the interrupt is accepted.

- **Bit 7: Stack pointer specifying flag (U flag)**

The user stack pointer (USP) is selected when this flag is 1. The interrupt stack pointer (ISP) is selected when the flag is 0.

This flag is cleared to 0 when a hardware interrupt is accepted or an INT instruction of software interrupt numbers 0 to 31 is executed.

- **Bits 8 to 11: Reserved.**

- **Bits 12 to 14: Processor interrupt priority level (IPL)**

The processor interrupt priority level (IPL) consists of three bits, for specification of up to eight processor interrupt priority levels from level 0 to level 7.

If the priority level of a requested interrupt is greater than the processor interrupt priority level(IPL), the interrupt is enabled.

- **Bit 15: Reserved.**

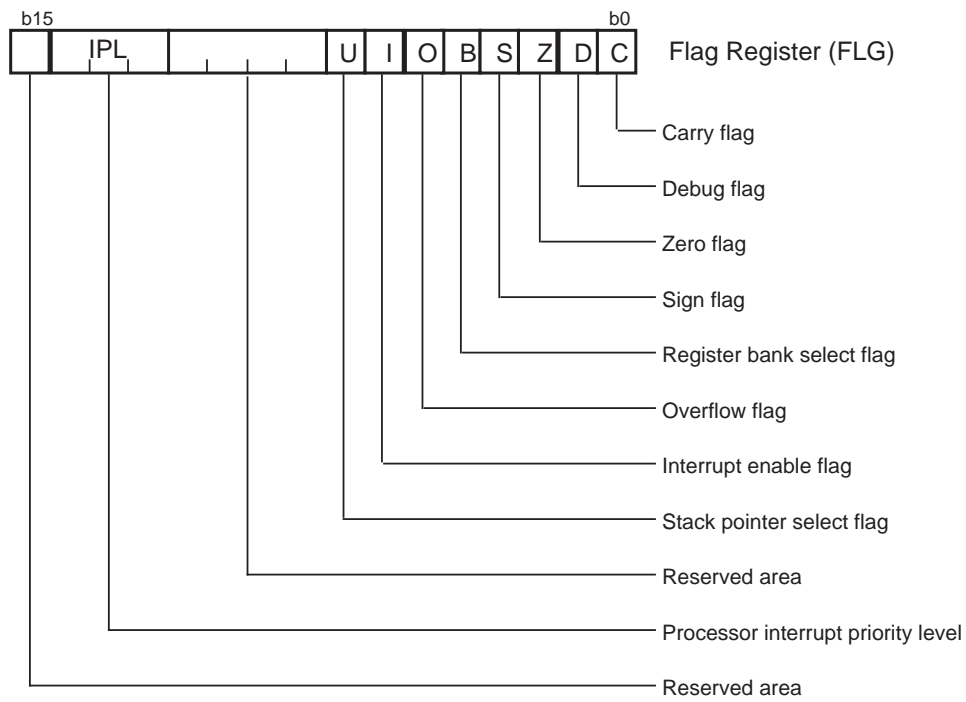


Figure 2.2.2 Bit configuration of flag register (FLG)

Register Status after Reset is Cleared

Table 2.2.1 lists the status of each register after a reset is cleared^(Note).

Table 2.2.1 Register Status after Reset Cleared

Register name	Status after a reset is cleared
Data register(R0/R1/R2/R3)	0000H
Address register(A0/A1)	000000H
Static base register(SB)	000000H
Flame base register(FB)	000000H
Interrupt table register(INTB)	000000H
User stack pointer(USP)	000000H
Interrupt stack pointer(ISP)	000000H
Flag register(FLG)	0000H
DMA mode register(DMD0/DMD1)	00H
DMA transfer count register(DCT0/DCT1)	Undefined.
DMA transfer count reload register(DRC0/DRC1)	Undefined.
DMA memory address register(DMA0/DMA1)	Undefined.
DMA SFR address register(DSA0/DSA1)	Undefined.
DMA memory address reload register(DRA0/DRA1)	Undefined.

Note: For the control register status in the SFR area after a reset is cleared, refer to the M16C/80 group data sheets and user's manuals.

2.3 Data Types

There are four data types handled by the M16C/80 series: integer, decimal (BCD), string, and bit. This section describes these data types.

Integer

An integer may be a signed or an unsigned integer. A negative value of a signed integer is represented by a 2's complement.

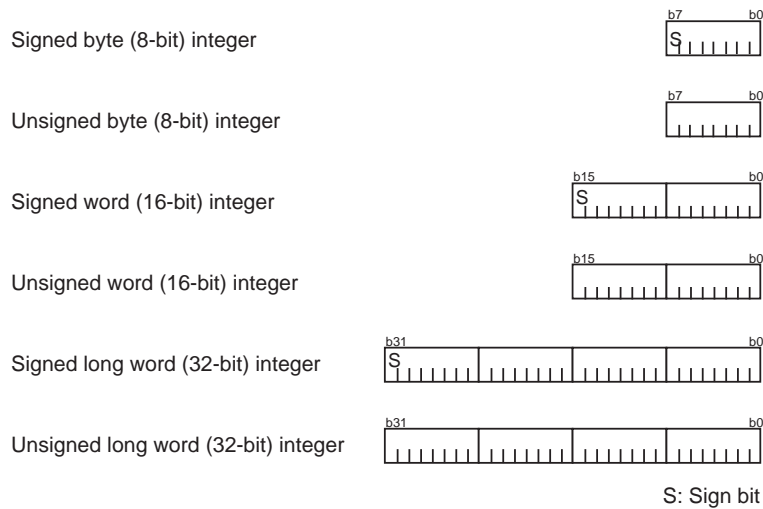


Figure 2.3.1 Integer data

Decimal (BCD)

The BCD code is handled in packed format.

This type of data can be used in four kinds of decimal arithmetic instructions: DADC, DADD, DSBB, and DSUB.

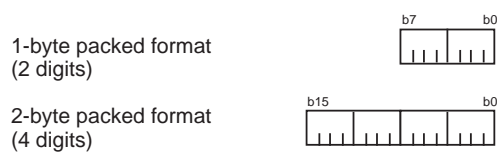
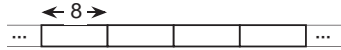


Figure 2.3.2 Decimal data

String

A string is a block of data comprised of a consecutive number of 1-byte or 1-word (16-bit) data. This type of data can be used in seven kinds of string instructions: SMOVB, SMOVF, SSTR, SCMPU, SIN and SOUT.

- String of byte (8-bit) data



- String of word (16-bit) data

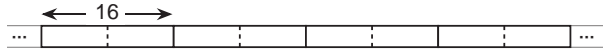


Figure 2.3.3 String data

Bit

Bit can be used in 14 kinds of bit instructions, including BCLR, BSET, BTST, and BNTST. Bits in each register are specified by a register name and a bit number, 0 to 15. Memory bits are specified by a different method in a different range depending on the addressing mode used. For details, refer to Section 2.5.4, "Bit Instruction Addressing".

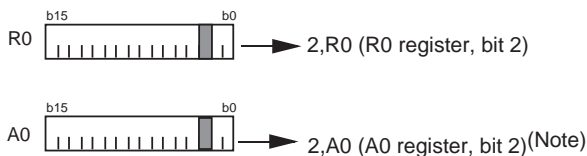


Figure 2.3.4 Specification of register bits

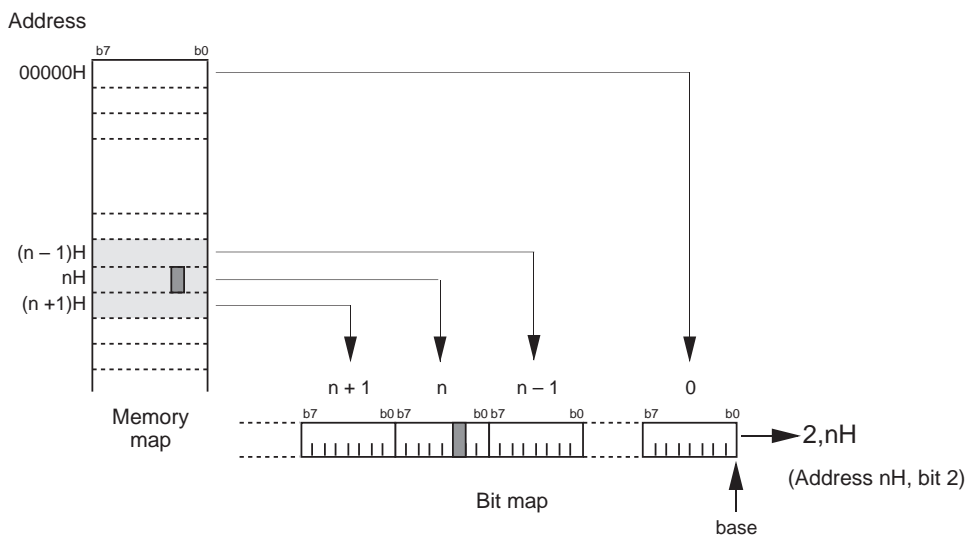


Figure 2.3.5 Specification of memory bits

Note : A0 and A1 register can be specified by the lower 8 bit.

2.4 Data Arrangement

The M16C/80 series can handle nibble (4-bit) and byte (8-bit) data efficiently. This section explains the data arrangements that can be handled by the M16C/80 series.

Data Arrangement in Register

Figure 2.4.1 shows the relationship between the data sizes and the bit numbers of a register. As shown below, the bit number of the least significant bit (LSB) is 0. The bit number of the most significant bit (MSB) varies with the data sizes handled.

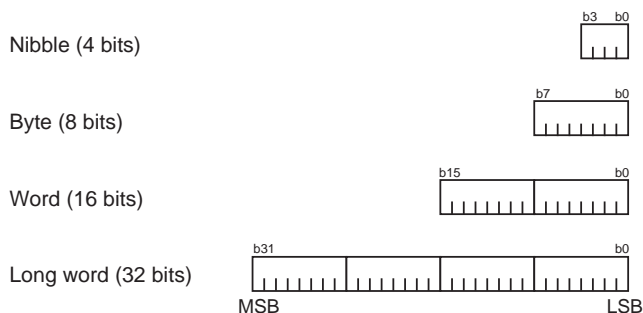


Figure 2.4.1 Data arrangement in register

Data Arrangement in Memory

Figure 2.4.2 shows the data arrangement in the M16C/80 series memory. Data is arranged in memory in units of 8 bits as shown below. A word (16 bits) is divided between the lower byte and the upper byte, with the lower byte, DATA(L), placed in a smaller address location. Similarly, addresses (24 bits) and long words (32 bits) are located in memory beginning with the lower byte, DATA(L) or DATA(LL).

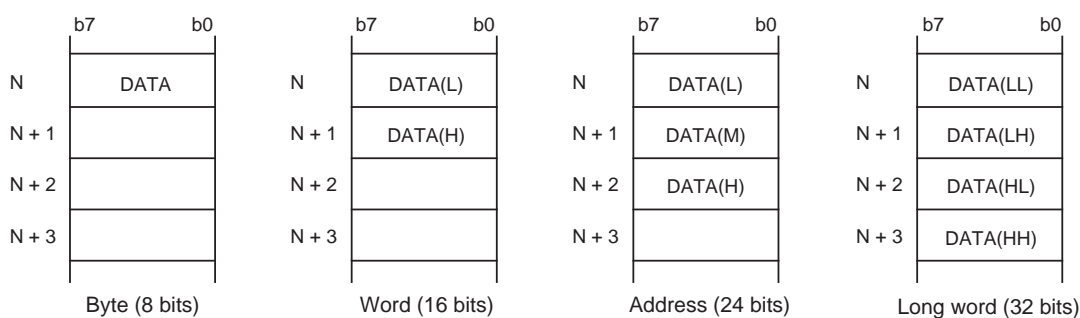


Figure 2.4.2 Data arrangement in memory

2.5 Addressing Modes

This section explains the M16C/80 series addressing.

The four types of addressing modes shown below are available.

(1) General instruction addressing

The entire address space from address 000000H to FFFFFFFH is accessed.

(2) Indirect instruction addressing

The entire address space from address 000000H to FFFFFFFH is accessed.

(3) Special instruction addressing

The entire address space from 000000H to FFFFFFFH is accessed and control registers.

(4) Bit instruction addressing

The entire address space from address 000000H to FFFFFFFH is accessed in units of bits.

List of Addressing Modes

All addressing modes are summarized in Table 2.5.1 and Table 2.5.2 below.

Table 2.5.1 Addressing Modes of M16C/80 Series 1

Item	Content	
Addressing mode	General instruction	Indirect instruction
Immediate	imm:8/16/32 bits	X
Register direct	Data register and address registers only	X
Control register direct	X	X
Absolute	abs:16 bits (0 to FFFFH) 24 bits (0 to FFFFFFFH)	X
Absolute indirect	X	[abs : 16/24 bits] (0 to FFFFFFFH)
Address register indirect	[A0] or [A1] without disp	X
Two-stage address register indirect	X	[[A0]] or [[A1]] without disp (0 to FFFFFFFH)
Address register relative	[A0] or [A1] dsp : 8/16/24 bits	X
Address register relative indirect	X	[dsp:8/16/24[A0]] or [dsp:8/16/24[A1]] (0 to FFFFFFFH)
SB relative and FB relative	dsp:8[SB] dsp:16[SB] dsp:8/16 bits(0 to 255 / 0 to 65534)	X
	dsp:8[FB] dsp:16[FB] dsp:8/16 bits(-128 to +127 / 32768 to +32767)	X
SB relative indirect and FB relative indirect	X	[dsp:8/16[SB]] (0 to FFFFFFFH)
	X	[dsp:8/16[FB]] (0 to FFFFFFFH)
Stack pointer relative	dsp:8[SP] dsp : 8 bits (-128 to +127) *MOV instruction only	X
Program counter relative	X	X
FLG direct	X	X

Table 2.5.2 Addressing Modes of M16C/80 Series 2

Item	Content	
Addressing mode	Special instruction	Bit instruction
Immediate	X	X
Register direct	X	R0L/R0H/R1L/R1H/A0/A1 only
Control register direct	INTB,ISP,SP,DMD0 etc.control register only	X
Absolute	X	base:19/27 bits (0 to FFFFH / 0 to 0FFFFFFFH)
Absolute indirect	X	X
Address register indirect	X	bit,[A0] or bit,[A1](0H to 0FFFFFFFH) bit:0 to 7
Two-stage address register indirect	X	X
Address register relative	X	bit,base[A0] or bit,base[A1] base:11/19/27
Address register relative indirect	X	X
SB relative and FB relative	X	bit,base:11[SB] (0H to FFH) bit,base:19[SB] (0H to FFFFH)
	X	bit,base:11[FB] (-128 to +127) bit,base:19[FB] (-32768 to +32767)
SB relative indirect and FB relative indirect	X	X
	X	X
Stack pointer relative	X	X
Program counter relative	label: .S: +0 to +7(JMP instruction only) .B: -128 to +127(JMP,JSR instruction only) .W: -32768 to +32767(JMP,JSR instruction only) .without length: -127 to +128(Jcn instruction only)	X
FLG direct	X	U, I, O, B, S, Z, D, C flag *FSET,FCLR instruction only

2.5.1 General Instruction Addressing

This section explains each addressing in the general instruction addressing mode.

Immediate

The immediate indicated by #IMM is the subject on which operation is performed. Add a # before the immediate.

Symbol: #IMM, #IMM8, #IMM16, #IMM32

Example: #123 (decimal)

#7DH (hexadecimal)

#01111011B (binary)

Absolute

The value indicated by abs16/24 is the effective address on which operation is performed. The range of effective addresses is 000000H to 000FFFFH at abs16, and 000000H to FFFFFFFH at abs24.

Symbol: abs16 or abs 24

Example: MOV.B #12H,DATA

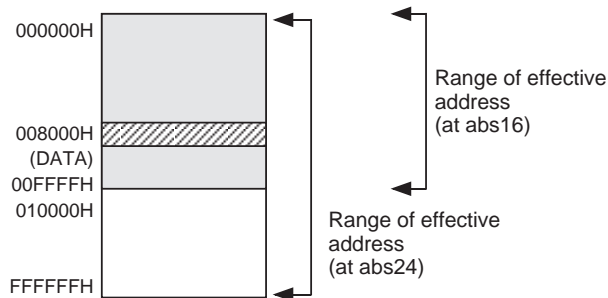


Figure 2.5.1 Absolute addressing

Register direct

A specified register is the subject on which operation is performed.

However, only the data and address registers can be used here.

Symbol: 8 bits R0L, R0H, R1L, R1H

16 bits R0, R1, R2, R3, A0, A1

32 bits R2R0, R3R1

Address Register Indirect

The value of an address register is the effective address to be operated on. The range of effective addresses is 000000H to FFFFFFFH.

Symbol: [A0], [A1]

Example: MOV.B #12H, [A0]

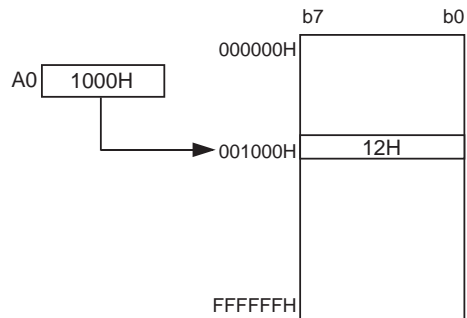


Figure 2.5.2 Address register indirect addressing

Address Register Relative

The value of an address register plus a displacement (dsp)^(Note) is the effective address to be operated on. The range of effective addresses is 000000H to FFFFFFFH. If the addition result exceeds FFFFFFFH, the most significant bits above and including bit 25 are ignored and the address returns to 000000H.

Symbol: dsp:8[A0], dsp:16[A0], dsp:24[A0], dsp:8[A1], dsp:16[A1], dsp:24[A1]

(1) When dsp is handled as a displacement

Example: MOV.B #34H,5[A0]

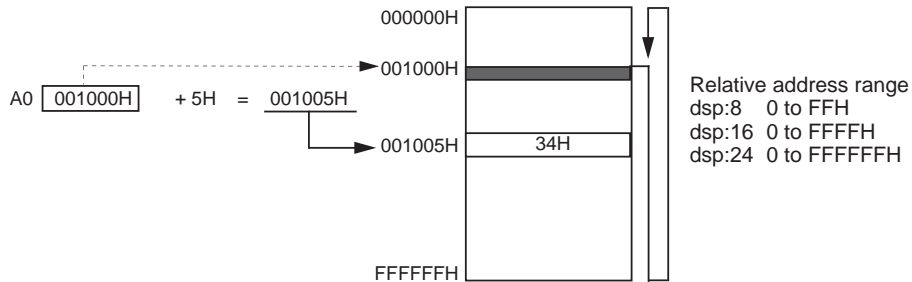


Figure 2.5.3 Address register relative addressing 1

(2) When address register (A0) is handled as a displacement

Example: MOV.B #56H,1234H[A0]

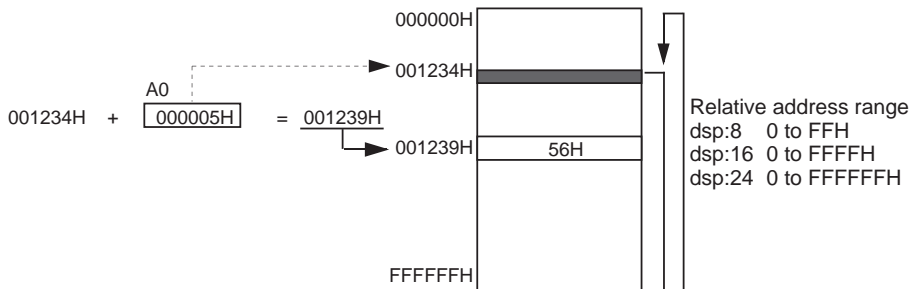


Figure 2.5.4 Address register relative addressing 2

(3) When the addition result exceeds 0FFFFFFH

Example: MOV.B #56H,1234H[A0]

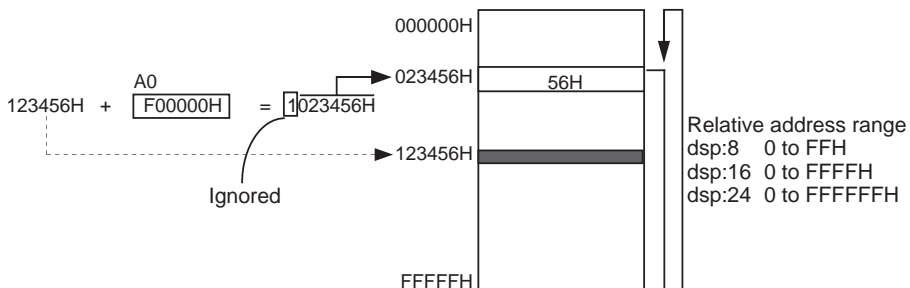


Figure 2.5.5 Address register relative addressing 3

Note: The displacement (dsp) refers to a displacement from the reference address. In this manual, 8-bit dsp is expressed as dsp:8, 16-bit dsp is expressed as dsp:16, and 24-bit dsp is expressed as dsp:24.

SB Relative

The address indicated by the content of static base register(SB) plus the value indicated by displacement(dsp) -added not including the sign bits- constitutes the effective address to be operated on. The range of effective addresses is 000000H to FFFFFFFH. However, if the addition resulted in exceeding FFFFFFFH ,the bits above bit 25 are ignored, and the address returns to 000000H.

Symbol: dsp:8[SB], dsp:16[SB]

Example: MOV.B #12H,5[SB]

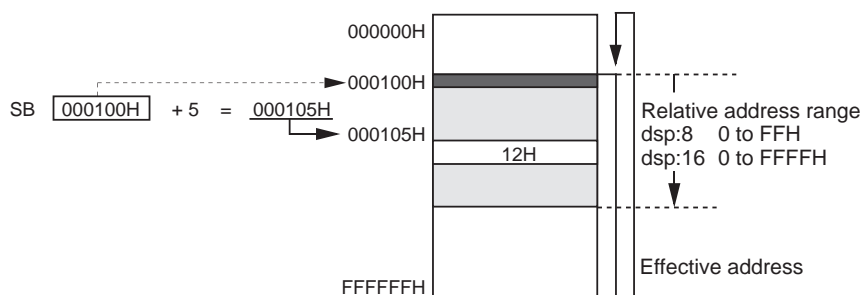


Figure 2.5.6 SB relative addressing

FB Relative

The address indicated by the content of frame base register(FB) plus the value indicated by displacement(dsp) -added not including the sign bits- constitutes the effective address to be operated on. The range of effective addresses is 000000H to FFFFFFFH. However, if the addition resulted in exceeding 000000H to FFFFFFFH ,the bits above bit 25 are ignored, and the address returns to 000000H or FFFFFFFH.

Symbol: dsp:8[FB]

(1) When dsp is a positive value

Example: MOV.B #12H,5[FB]

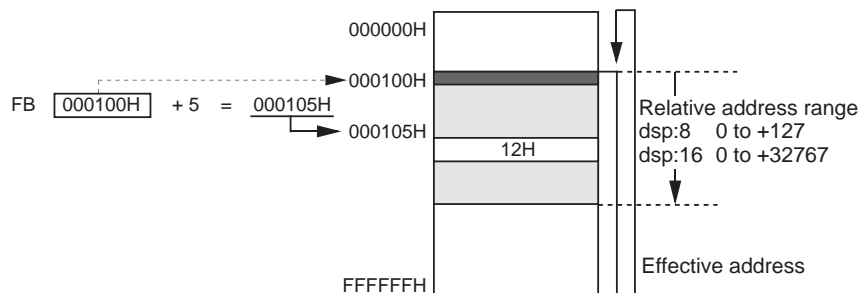


Figure 2.5.7 FB relative addressing 1

(2) When dsp is a negative value

Example: MOV.B #12H,-5[FB]

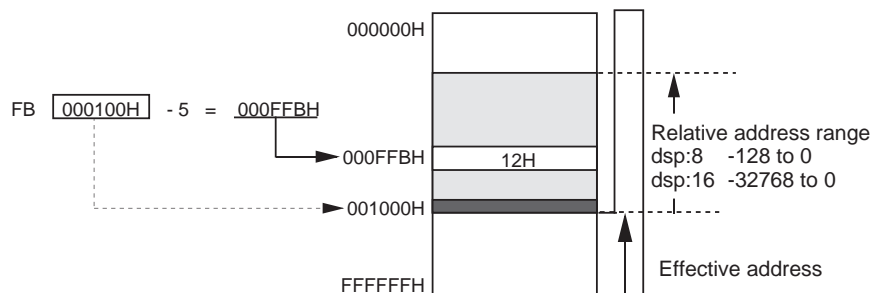


Figure 2.5.8 FB relative addressing 2

Column Difference between SB Relative and FB Relative

In SB relative addressing, the address indicated by the SB register content and the value indicated by dsp are added not including the sign and the result of addition is the effective address to be operated on. The relative range is 0 to +255 (FFH) for dsp: 8[SB], and 0 to +65535 (FFFFH) for dsp: 16[SB].

In FB relative addressing, dsp is added to or subtracted from the address indicated by the FB register content and the result of addition or subtraction is the effective address to be operated on. The relative range is -128 to +127 (80H to 7FH) for dsp: 8[FB], and -32768 to +32767 (8000H to 7FFFH) for dsp: 16[FB]. FB relative allows accessing memory locations in the negative direction. The dsp used for this addressing can be 8 bits or 16 bits.

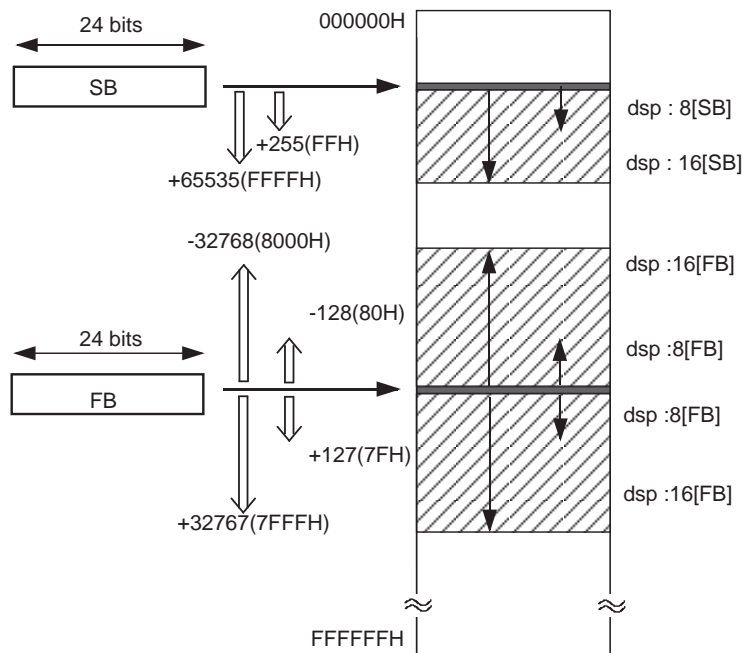


Figure 2.5.9 SB relative and FB relative addressing

Column Application Example of SB Relative

SB relative addressing can be applied for the specific data table of each subroutine as shown in Figure 2.5.10. Although the data necessary to run each subroutine must be switched over when calling the subroutine, use of SB relative addressing helps to accomplish this switchover by only rewriting the SB register.

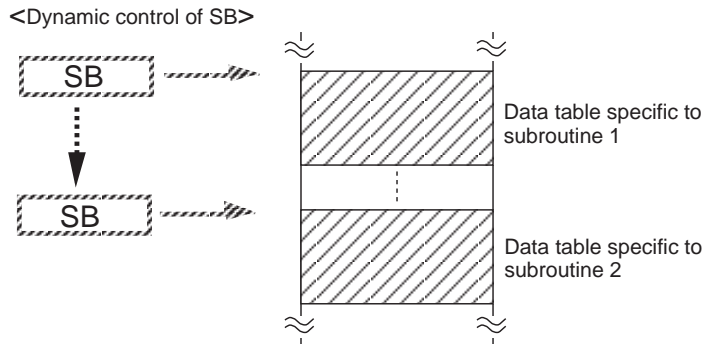


Figure 2.5.10 Application example of SB relative addressing

Column Application Example of FB Relative

FB relative addressing can be used for the stack frame that is created when calling a function, as shown in Figure 2.5.11. Since the local variable area in the stack frame is located in the negative direction of addresses, FB relative addressing is needed because it allows for access in both positive and negative directions from the base.

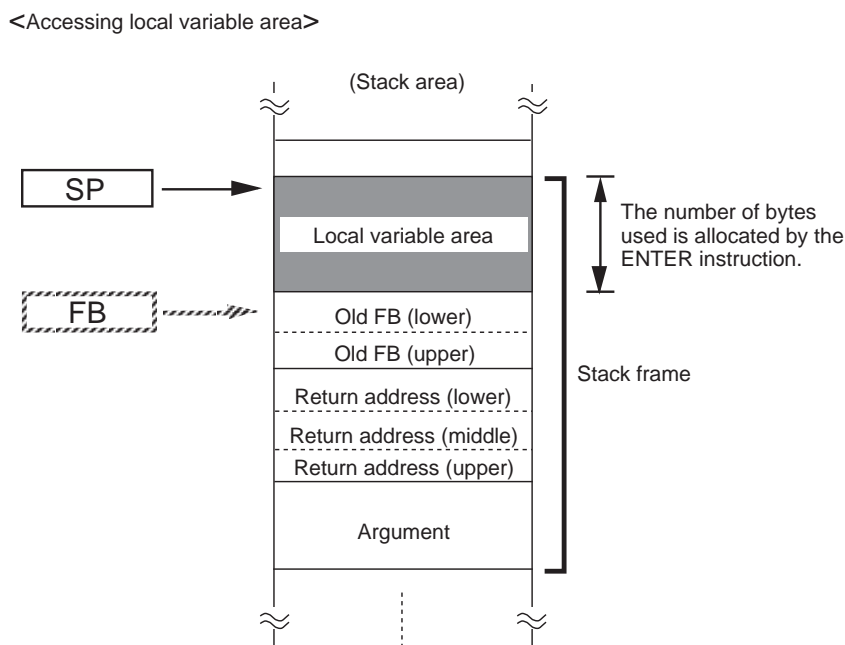


Figure 2.5.11 Application example of FB relative addressing

Stack Pointer Relative (SP Relative)

In SP relative addressing, the address indicated by the SB register content and the value indicated by dsp are added including the sign and the result of addition is the effective address to be operated on. SP relative addressing can only be used in the MOV instruction. The range of effective addresses is 000000H to FFFFFFFH. If the result of addition exceeds the range of 000000H to FFFFFFFH, any value above 25 bits is ignored and the address wraps around to 000000H or FFFFFFFH.

Symbol: dsp:8[SP]

(1) When dsp is a positive value

Example: MOV.B R0L,5[SP]

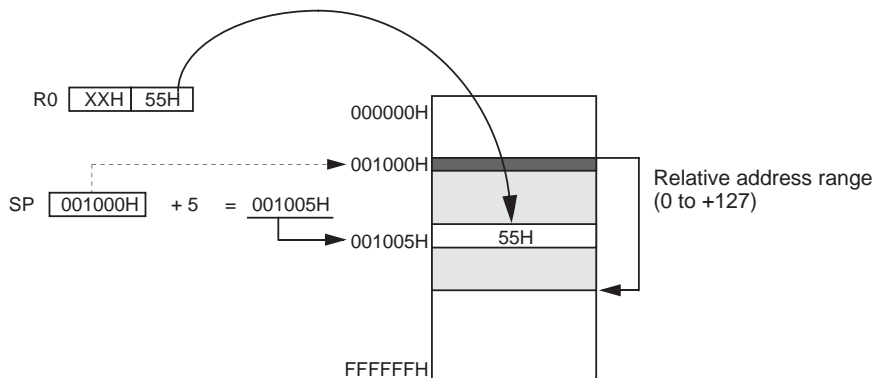


Figure 2.5.12 SP relative addressing 1

(2) When dsp is a negative value

Example: MOV.B R0L,-5[SP]

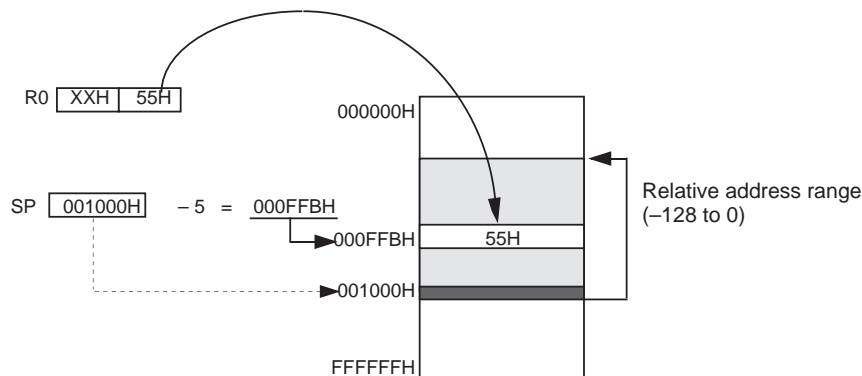


Figure 2.5.13 SP relative addressing 2

Column Relative Address Ranges of Relative Addressing

The relative address ranges of relative addressing are summarized in Table 2.5.3.

Table 2.5.3 Relative Address Ranges of Relative Addressing

Addressing mode	Descriptive form	Relative range
Address register relative	dsp:8[An] dsp:16[An] dsp:24[An]	0 to 255(FFH) 0 to 65535(FFFFH) 0 to 16777215(FFFFFFH)
SB relative and FB relative	dsp:8[SB] dsp:16[SB] dsp:8[FB] dsp:16[FB]	0 to 255(0FFH) 0 to 65535(0FFFFH) -128(80H) to +127(7FH) -32768(8000H) to +32767(7FFFFH)
Stack pointer relative	dsp:8[SP]	-128(80H) to +127(7FH)

2.5.2 Indirect instruction Addressing

The Indirect instruction addressing accesses an area from address 000000H to FFFFFFFH. This section explains each addressing in the indirect instruction addressing mode.

Absolute indirect

The 4-bytes value indicated by absolute addressing constitutes the effective address to be operated on. The effective address range is 000000H to FFFFFFFH.

Symbol: [abs16] or [abs24]

Example: MOV.B [001000H],R0L

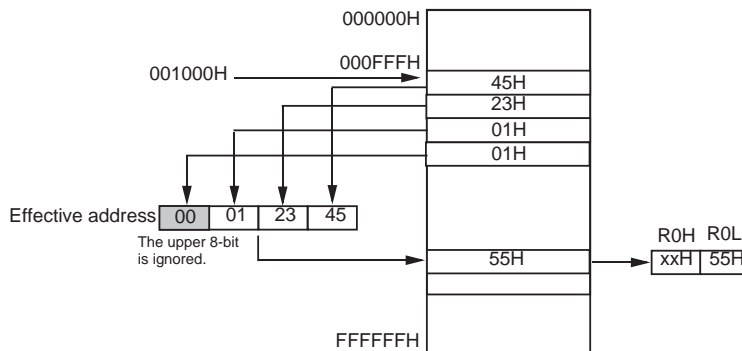


Figure 2.5.14 Absolute indirect addressing

Two-stage address register indirect

The 4-bytes value indicated by address register(A0/A1) indirect constitutes the effective address to be operated on. The effective address range is 000000H to FFFFFFFH.

Symbol: [[A0]] or [[A1]]

Example: MOV.B [[A0]],R0L

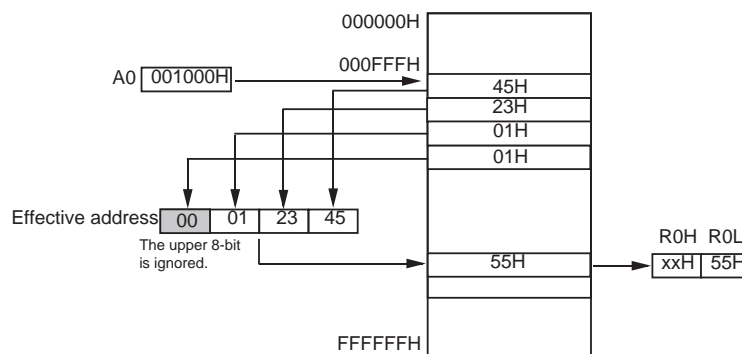


Figure 2.5.15 Two-stage address register indirect addressing

Address register relative indirect

The 4-bytes value indicated by address register relative constitutes the effective address to be operated on. The effective address range is 000000H to FFFFFFFH.

Symbol: [dsp:8[A0]], [dsp:8[A1]], [dsp:16[A0]], [dsp:16[A1]], [dsp:24[A0]], or [dsp:24[A1]]

Example: MOV.B [5[A0]], R0L

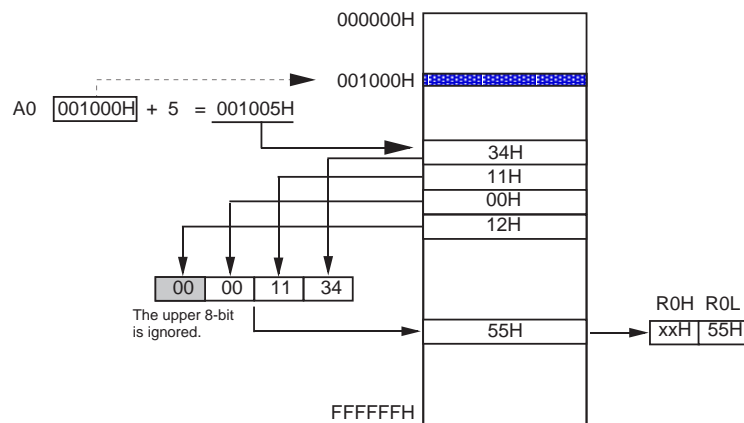


Figure 2.5.16 Address register relative indirect addressing

SB relative indirect

The 4-byte value indicated by SB relative constitutes the effective address to be operated on. The effective address range is 000000H to FFFFFFFH.

Symbol: [dsp:8[SB]] or [dsp:16[SB]]

Example: MOV.B [2[SB]], R0L

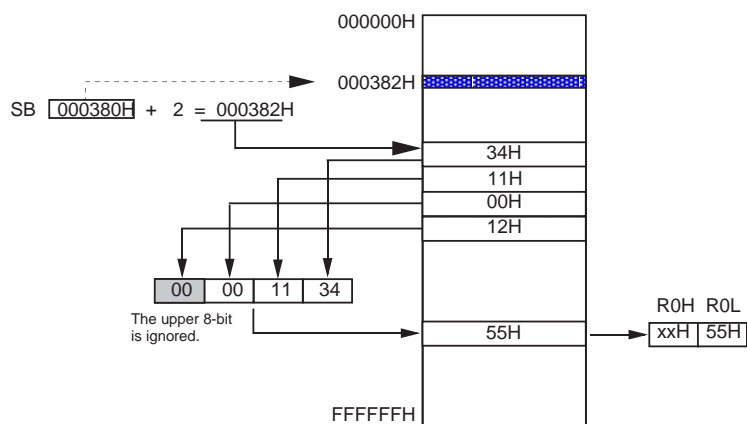


Figure 2.5.17 SB relative indirect addressing

FB relative indirect

The 4-byte value indicated by FB relative constitutes the effective address to be operated on. The effective address range is 000000H to FFFFFFFH.

Symbol: [dsp:8[FB]] or [dsp:16[FB]]

Example: MOV.B [2[FB]],R0L

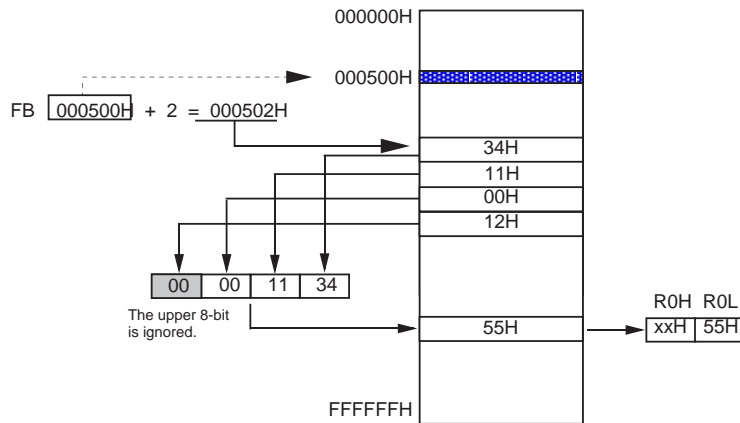


Figure 2.5.18 FB relative indirect addressing 1

Example: MOV.B [-25[FB]],R0L

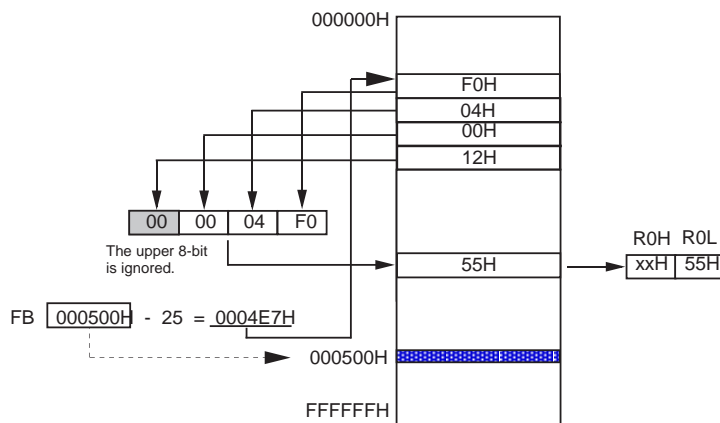


Figure 2.5.19 FB relative indirect addressing 2

2.5.3 Special Instruction Addressing

The Special Instruction addressing accesses an area from address 000000H to FFFFFFFH and control registers. This section explains each addressing in the special instruction addressing mode.

Control register direct

The specified control register is the object to be operated on. This addressing can be used in LDC, STC, POPC, and PUSHC instructions.

If you specify SP, the stack pointer indicated by the U flag is the object to be operated on.

Symbol: INTB, ISP, SP, SB, FB, FLG, SVP, VCT, SVF, DMD0, DMD1, DCT0, DCT1, DRC0, DRC1, DMA0, DMA1, DSA0, DSA1, DRA0, or DRA1

Example: LDC #001000H, ISP



Figure 2.5.20 Control register direct addressing

Program counter relative

When the jump length specifier (.length) is (.S) ... the base address plus the value indicated by displacement(dsp) -added not including the sign bit- constitutes the effective address.

This addressing can be used in JMP instruction.

(1)When the jump length specifier (.length) is (.S)

Symbol:label ($PC+2 \leq \text{label} \leq PC+9$)

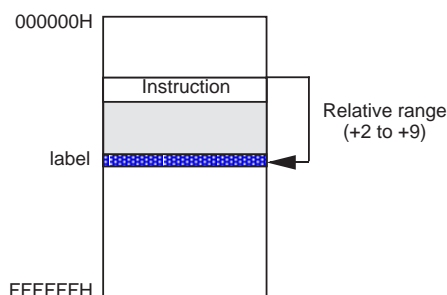


Figure 2.5.21 Program counter relative addressing 1

When the jump length specifier (.length) is (.B) or (.W) ... the base address plus the value indicated by displacement(dsp) -added including the sign bits- constitutes the effective address.

This addressing can be used in JMP and JSR instructions.

(2)When the jump length specifier (.length) is (.B)

Symbol:label ($PC-128 \leq \text{label} \leq PC+127$)

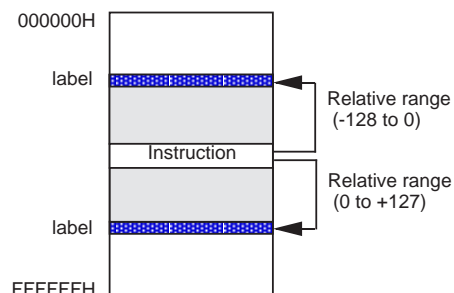
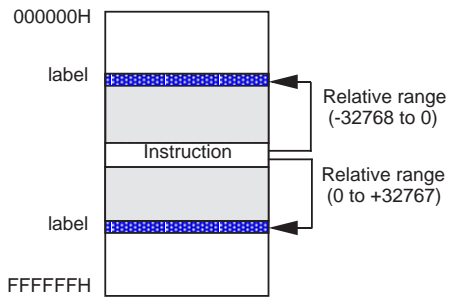


Figure 2.5.22 Program counter relative addressing 2

(3)When the jump length specifier (.length) is (.W)Symbol:label ($PC-32768 \leq \text{label} \leq PC+32767$)**Figure 2.5.23 Program counter relative addressing 3**

If the jump distance specifier(.length) is omitted ,the assembler chooses the optimum specifier. And if the addition resulted in exceeding 000000H to FFFFFFFH the bits above bit 25 are ignored,and the address returns to 000000H or FFFFFFFH.

2.5.4 Bit Instruction Addressing

The Bit Instruction addressing accesses an area from address 000000H to FFFFFFFH . This addressing can be used in the bit instructions. This section explains each addressing in the bit instruction addressing mode.

Absolute

The bit that is as much away from bit 0 at the address indicated by base as the number of bits indicated by bit is the object to be operated on.

The address range that can be specified by bit ,base:19 and bit,base:27 respectively are 000000H to 00FFFFH and 000000H to FFFFFFFH.

Symbol: bit,base:19 or bit,base:27

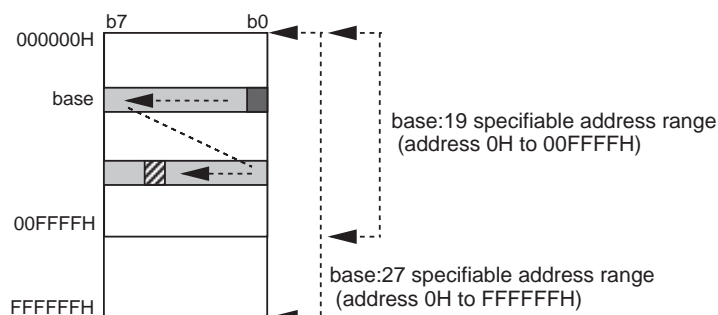


Figure 2.5.24 Bit instruction absolute addressing 1

Example 1: BCLR 18,base_addr

Example 2: BCLR 4,base_addr2

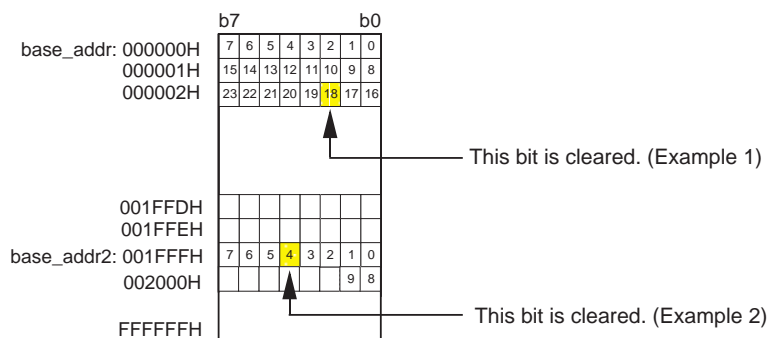


Figure 2.5.25 Bit instruction absolute addressing 2

Register direct

The specified register bit is the object to be operated on. For the bit position(bit) you can specify 0 to 7. For the address register(A0,A1), you can specify 8 low-order bits.

Symbol: bit,R0L, bit,R0H, bit,R1L, bit,R1H, bit,A0, or bit,A1

Example: BCLR 6,R0L

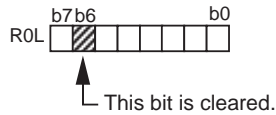


Figure 2.5.26 Bit instruction register direct addressing

FLG Direct

The specified flag is the object to be operated on. This addressing can be used in FCLR and FSET instructions. The bit positions that can be specified here are only the 8 low-order bits of the FLG register.

Symbol: U, I, O, B, S, Z, D, C

Example: FSET U

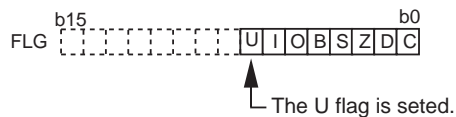


Figure 2.5.27 Bit instruction FLG direct addressing

Address Register Indirect

The bit that is as much away from bit 0 at address indicated by address register(A0/A1) as the number of bits is the object to be operated on.
 Bits at addresses 000000H to FFFFFFFH can be the object to be operated on. For the bit position (bit) you can specify 0 to 7.
 Symbol: bit,[A0] or bit,[A1]
 Example: BCLR 5,[A0]

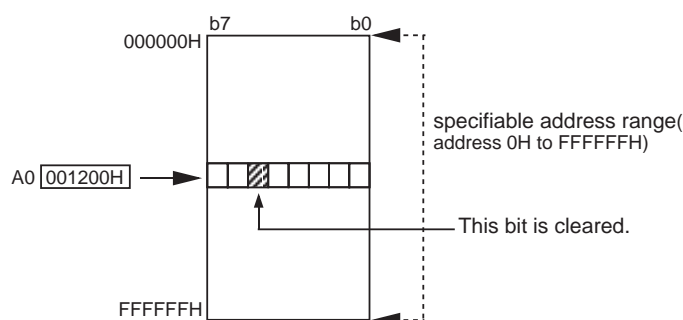


Figure 2.5.28 Bit instruction address register indirect addressing

Address Register Relative

The bit that is as much away from bit 0 at the address indicated by base -added not including the sign bits- as the number of bits indicated by address register(A0/A1) is the object to be operated on. The effective address range is 000000H to FFFFFFFH. However, if the address of the bit to be operated on exceeds FFFFFFFH, the bits above bit 25 are ignored and address returns 000000H. The address range that can be specified by bit,base:11, bit,base:19, and bit,base:27 respectively are 256 bytes, 65536 bytes, and 16777216 bytes from address register(A0/A1) value.
 Symbol: bit,base:11[A0], bit,base:11[A1], bit,base:19[A0], bit,base:19[A1], bit,base:27[A0], or bit,base:27[A1]
 Example: BCLR 5,26H[A0]

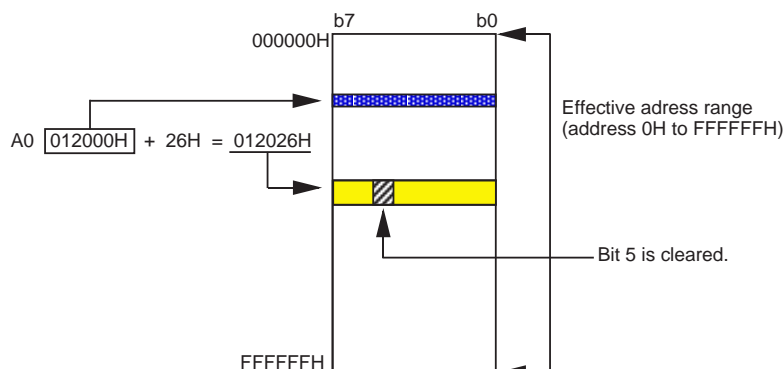


Figure 2.5.29 Bit instruction address register relative addressing

SB Relative

In this mode, the address is referenced to the value indicated by the SB register. The value of the SB register has base added without a sign. The resulting value indicates the reference address, so operation is performed on the bit that is away from bit 0 at that address by a number of bits indicated by bit.

The address range that can be specified by bit,base:11, and bit,base:19 respectively are 256 bytes, and 65536 bytes form the static base register(SB) value. However, if the address of the bit to be operated on exceeds FFFFFFFH, the bits above bit 25 are ignored and the address returns to 000000H.

Symbol: bit,base:11[SB] or bit,base:19[SB]

Note: bit,base:11 [SB] : One bit in an area of up to 256 bytes can be specified.

bit,base:19 [SB] : One bit in an area of up to 64 K bytes can be specified.

Example: BCLR 13,8[SB]

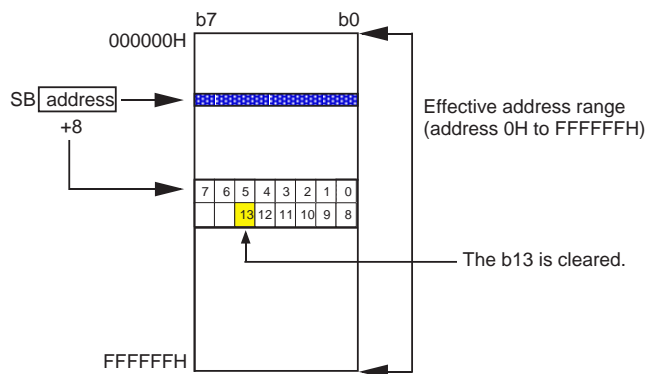


Figure 2.5.30 Bit instruction SB relative addressing

FB Relative

In this mode, the address is referenced to the value indicated by the FB register. The value of the FB register has base added with the sign included. The resulting value indicates the reference address, so operation is performed on the bit that is away from bit 0 at that address by a number of bits indicated by bit.

Symbol: bit,base:11[FB] or bit,base:19[FB]

Example: BCLR 5,-8[FB]

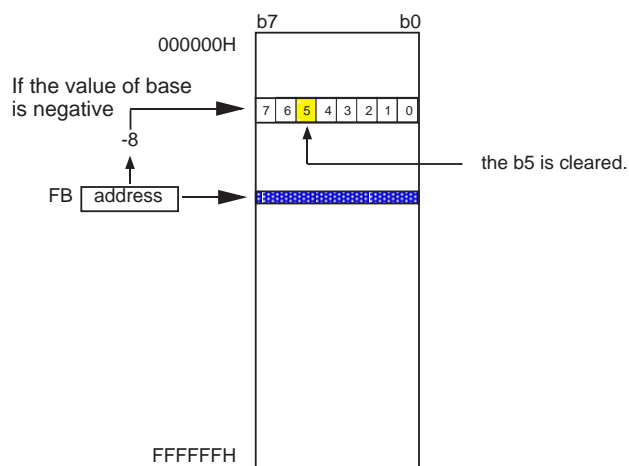


Figure 2.5.31 Bit instruction FB relative addressing

2.6 Instruction Set

This section explains the instruction set of the M16C/80 series. The instruction set is summarized by function in list form. In addition, some characteristic instructions among the instruction set are explained in detail.

The table below shows the symbols used in the list and explains their meanings.

Symbol	Meaning
src	Operand that does not store processing result.
dest	Operand that stores processing result.
label	Operand that means an address.
abs16/24	Absolute value.(16 bits or 24 bits)
abs20	20-bit absolute value.
dsp:8/16/24	Displacement.(8 bits, 16 bits, or 24 bits)
dsp:16	16-bit displacement.
#IMM/4/8/16/24/32	Immediate value.(8 bits, 16 bits, 24 bits, or 32 bits)
.size	Size specifier.(.B or .W)
.length	Jump distance specifier.(.S, .B, .W, or .A)
←	Transfers in the direction of arrow.
+	Add.
-	Subtract.
*	Multiply.
/	Divide.
&	Logical AND.
	Logical OR.
^	Exclusive OR.
—	Negate.
	Absolute value.
EXT()	Extend sign in ().
U,I,O,B,S,Z,D,C	Flag name.
R0L,R0H,R1,R1H	8-bit register name.
R0,R1,R2,R3,A0,A1	16-bit register name.
R2R0,R3R1,A1A0	32-bit register name.
SB,FB,SP,PC	Register name.
MOV <i>Dir</i> ,BM <i>Cnd</i> ,JC <i>Cnd</i>	Dir(direction) and Cnd(condition) mnemonics are shown in italic.
JGEU/C,JEQ/Z	Indecate that JGEU/C is written as JGEU or JC, and that JEQ/Z is wrriten as JEQ or JZ.
INDEX <i>type</i>	Mnemonics of type(modifier type) are shown in italic.
"O"	(Addressing) Can be used.
	(Flag change) Flag changes according to execution result.
"_"	(Flag change) Flag does not change.

2.6.1 Instruction Description

This section explains the format in which M16C/80 instructions are written.

Format of instruction description

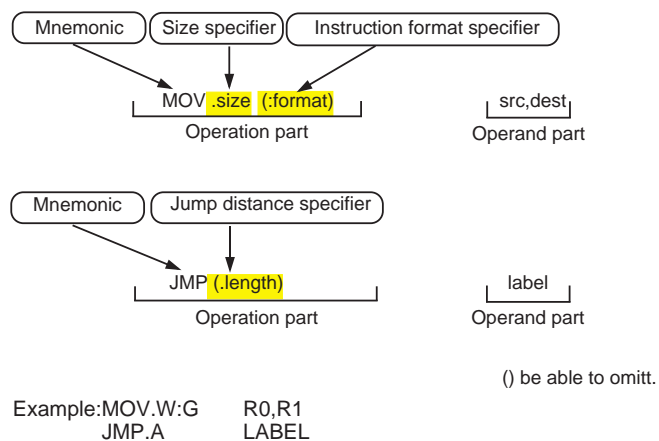


Figure 2.6.1 Format of instruction description

- Mnemonic : Indicates the operation performed by the instruction.
- Size specifier : Specifies the data size to be operated on by the mnemonic.
- Branch distance specifier : Specifies the distance to the target address of a branch instruction or subroutine call instruction. (Normally omitted.)
- Format specifier : Specifies the format of the op-code. The code lengths of the op-code and operand vary with the op-code format. (Normally omitted.)

Specifiers

Size specifier	Content
.B	Specifies byte size. (8 bits)
.W	Specifies word size. (16 bits)
.L	Specifies long word size (32 bits)

Branch distance specifier	Content
.S	Branch distance: +2 to +9 (3-bit forward relative)
.B	Branch distance: -128 to +127 (8-bit relative)
.W	Branch distance: -32768 to +32767 (16-bit relative)
.A	Branch distance: 0 to FFFFFFFH (24-bit absolute)

Instruction format specifier ^(Note)	Content	Selection priority
.Z	Zero format	High
.S	Short format	
.Q	Quick format	
.G	Generic format	Low

Note: Some instructions do not have the instruction format specifier.

Figure 2.6.2 Specifiers

Instruction format**1.Generic format(:G)**

The op-code includes information on the operation to be performed, as well as src and dest addressing information.

Table 2.6.1 Generic format

Op-code	Src code	Dest code
2 to 3 bytes	0 to 4 bytes	0 to 3 bytes

2.Quick format(:Q)

The op-code includes information on the operation to be performed and the immediate data, as well as dest addressing information. However, the immediate data included in the op-code is a value that can be expressed by -7 to +8 or -8 to +7 (varying with the instruction).

Table 2.6.2 Quick format

Op-code	Dest code
2 bytes	0 to 3 bytes

3.Short format(:S)

The op-code includes information on the operation to be performed, as well as src and dest addressing information. However, the usable addressing modes are limited. The S format can be used in part of addressing modes.

Table 2.6.3 Short format

Op-code	Src code	Dest code
1 byte	0 to 2 bytes	0 to 2 bytes

4.Zero format(:Z)

The op-code includes information on the operation to be performed and the immediate data, as well as dest addressing information. However, the immediate data is fixed to 0. The Z format can be used in part of addressing modes.

Table 2.6.4 Zero format

Op-code	Dest code
1 byte	0 to 2 bytes

2.6.2 Instruction List

In this and following pages, instructions are summarized by function in list form, showing the content of each mnemonic, addressing, and flag changes.

Transfer

Mnemonic		Explanation
MOV.size ^(Note)	src,dest	Transfers src to dest or sets immediate in dest.
MOVA	src,dest	Transfers address in src to dest.
MOVHH	src,dest	Transfers 4 high-order bits in src to 4 high-order bits in dest.
MOVHL	src,dest	Transfers 4 high-order bits in src to 4 low-order bits in dest.
MOVLH	src,dest	Transfers 4 low-order bits in src to 4 high-order bits in dest.
MOVLL	src,dest	Transfers 4 low-order bits in src to 4 low-order bits in dest.
MOVX	src,dest	Sign-extends 8-bit immediate value to 32 bits before transferring to dest.
POP.size	dest	Restores value from stack area.
POPC	dest	Restores value from stack area to dedicated register indicated by dest.
POPM	dest	Restores multiple registers values collectively from stack area.
PUSH.size	src	Saves register / memory / immediate to stack area.
PUSHA	src	Saves address in src to stack area.
PUSHC	src	Saves dedicated src register to stack area.
PUSHM	src	Saves multiple registers to stack area.
SIN.size		Transfers string in address incrementing direction using A0 as fixed source address of transfer, A1 as destination address of transfer, and R3 as transfer count.
SMOVB.size		Transfers string in address decrementing direction using A0 as source address of transfer, A1 as destination address of transfer, and R3 as transfer count.
SMOVF.size		Transfers string in address incrementing direction using A0 as source address of transfer, A1 as destination address of transfer, and R3 as transfer count.
SMOVU.size		Transfers string in address incrementing direction using A0 as source address of transfer, A1 as destination address of transfer, and R3 as transfer count until 0 is detected.

Note: Write .W or .B for .size.

Operand	Addressing										Flag change							
	General instruction									Special instruction								
	Immediate	Absolute	Register direct	Address register indirect	Address register relative	SB relative	FB relative	Stack pointer relative	Control register direct	Program counter relative	U	I	O	B	S	Z	D	C
src ^{*f}	O	O	O	O	O	O	O	O			-	-	-	-	O	O	-	-
dest ^f		O	O	O	O	O	O	O			-	-	-	-	O	O	-	-
src		O			O	O	O				-	-	-	-	-	-	-	-
dest			O								-	-	-	-	-	-	-	-
src			ROL ^{*a}								-	-	-	-	-	-	-	-
dest		O	O ^b	O	O	O	O				-	-	-	-	-	-	-	-
src		O	O ^b	O	O	O	O				-	-	-	-	-	-	-	-
dest			ROL ^{*a}								-	-	-	-	-	-	-	-
src	O ^{*c}										-	-	-	-	O	O	-	-
dest ^f		O	O	O	O	O	O				-	-	-	-	-	-	-	-
dest ^f		O	O	O	O	O	O				-	-	-	-	-	-	-	-
dest									O ^{*d}		*e	*e	*e	*e	*e	*e	*e	*e
dest									O		-	-	-	-	-	-	-	-
src ^{*f}	O	O	O	O	O	O	O				-	-	-	-	-	-	-	-
src		O			O	O	O				-	-	-	-	-	-	-	-
src									O ^{*d}		-	-	-	-	-	-	-	-
src			O						O		-	-	-	-	-	-	-	-
											-	-	-	-	-	-	-	-
											-	-	-	-	-	-	-	-
											-	-	-	-	-	-	-	-
											-	-	-	-	-	-	-	-

*a R0L register is selected for src or dest..

*b Can be selected from R0L, R0H, R1L, or R1H.

*c The immediate value is 8 bits in size, whose possible value is $-128 < \text{#IMM8} < +127$.

*d When SP is specified, the stack pointer indicated by the U flag is the target.

*e It is only when FLG is specified that dest changes.

*f Indirect addressing [src] and [dest] can be used in any register other than R0L/R0/R2R0, R0H/R2/-, R1H/R3/-, SP/SP/SP, dsp:8[SP], and #IMM.

Mnemonic	Explanation
SOUT.size ^(Note)	Transfers string in address incrementing direction using A0 as source address of transfer, A1 as fixed destination address of transfer, and R3 as transfer count.
SSTR.size	Transfers string in address incrementing direction using R0L/R0 as store data, A1 as destination address of transfer, and R3 as transfer count.
STNZ.size src,dest	Transfers src to dest when Z flag = 0.
STZ.size src,dest	Transfers src to dest when Z flag = 1.
STZX.size src1,src2,dest	Transfers src1 to dest when Z flag = 1 or src2 to dest when Z flag = 0.
XCHG.size src,dest	Exchanges contents of src and dest with each other.

Note: Write .W or .B for .size.

Addressing											Flag change							
Operand	General instruction									Special instruction	U	I	O	B	S	Z	D	C
	Immediate	Absolute	Register direct	Address register indirect	Address register relative	SB relative	FB relative	Stack pointer relative	Control register direct	Program counter relative								
											-	-	-	-	-	-	-	-
											-	-	-	-	-	-	-	-
src	O ^g										-	-	-	-	-	-	-	-
dest ^h		O	O	O	O	O	O				-	-	-	-	-	-	-	-
src	O ^g										-	-	-	-	-	-	-	-
dest ^h		O	O	O	O	O	O				-	-	-	-	-	-	-	-
src	O										-	-	-	-	-	-	-	-
dest ^h		O	O	O	O	O	O				-	-	-	-	-	-	-	-
src			O								-	-	-	-	-	-	-	-
dest ^h		O	O	O	O	O	O				-	-	-	-	-	-	-	-

*g The immediate value is selected for 8/16 bits in size.

*h Indirect addressing [dest] can be used in any register other than R0L/R0/R2R0, R0H/R2/-, R1H/R3/-, SP/SP/SP, dsp:8[SP], and #IMM.

Bit Manipulation

Mnemonic		Explanation
BAND	src	$C \text{ flag} \leftarrow \text{src} \ \& \ C \text{ flag}$; ANDs bits.
BCLR	dest	$\text{dest} \leftarrow 0$; Clears bits.
BITINDEX.size	src	Operand specified by src becomes the src or dest index value of the next bit instruction.
BMGEU/C	dest	If $C=1$, $\text{dest} \leftarrow 1$; otherwise, $\text{dest} \leftarrow 0$; Conditionally transfers bit.
BMLTU/NC	dest	If $C=0$, $\text{dest} \leftarrow 1$; otherwise, $\text{dest} \leftarrow 0$
BMEQ/Z	dest	If $Z=1$, $\text{dest} \leftarrow 1$; otherwise, $\text{dest} \leftarrow 0$
BMNE/NZ	dest	If $Z=0$, $\text{dest} \leftarrow 1$; otherwise, $\text{dest} \leftarrow 0$
BMGTU	dest	If $C \ \& \ Z=1$, $\text{dest} \leftarrow 1$; otherwise, $\text{dest} \leftarrow 0$
BMLEU	dest	If $C \ \& \ Z=0$, $\text{dest} \leftarrow 1$; otherwise, $\text{dest} \leftarrow 0$
BMPZ	dest	If $S=0$, $\text{dest} \leftarrow 1$; otherwise, $\text{dest} \leftarrow 0$
BMN	dest	If $S=1$, $\text{dest} \leftarrow 1$; otherwise, $\text{dest} \leftarrow 0$
BMGE	dest	If $S \ \vee \ O=0$, $\text{dest} \leftarrow 1$; otherwise, $\text{dest} \leftarrow 0$
BMLE	dest	If $(S \ \vee \ O) \ \vee \ Z=1$, $\text{dest} \leftarrow 1$; otherwise, $\text{dest} \leftarrow 0$
BMGT	dest	If $(S \ \vee \ O) \ \vee \ Z=0$, $\text{dest} \leftarrow 1$; otherwise, $\text{dest} \leftarrow 0$
BMLT	dest	If $S \ \vee \ O=1$, $\text{dest} \leftarrow 1$; otherwise, $\text{dest} \leftarrow 0$
BMO	dest	If $O=1$, $\text{dest} \leftarrow 1$; otherwise, $\text{dest} \leftarrow 0$
BMNO	dest	If $O=0$, $\text{dest} \leftarrow 1$; otherwise, $\text{dest} \leftarrow 0$
BNAND	src	$C \text{ flag} \leftarrow \text{src} \ \wedge \ C \text{ flag}$; ANDs inverted bits.
BNOR	src	$C \text{ flag} \leftarrow \text{src} \ \vee \ C \text{ flag}$; ORs inverted bits.
BNOT	dest	Inverts dest and stores in dest. ; Inverts bit.
BNTST	src	$Z \text{ flag} \leftarrow \text{src}$, $C \text{ flag} \leftarrow \text{src}$; Tests inverted bit.
BNXOR	src	$C \text{ flag} \leftarrow \text{src} \ \vee \ C \text{ flag}$; Exclusive ORs inverted bits.
BOR	src	$C \text{ flag} \leftarrow \text{src} \ \vee \ C \text{ flag}$;ORs bits.
BSET	dest	$\text{dest} \leftarrow 1$;Sets bit.
BTST	src	$Z \text{ flag} \leftarrow \text{src}$, $C \text{ flag} \leftarrow \text{src}$;Tests bit.
BTSTC	dest	$Z \text{ flag} \leftarrow \text{dest}$, $C \text{ flag} \leftarrow \text{dest}$, $\text{dest} \leftarrow 0$; Tests and clears bit.
BTSTS	dest	$Z \text{ flag} \leftarrow \text{dest}$, $C \text{ flag} \leftarrow \text{dest}$, $\text{dest} \leftarrow 1$; Tests and sets bit.
BXOR	src	$C \text{ flag} \leftarrow \text{src} \ \vee \ C \text{ flag}$; Exclusive ORs bits.

Addressing								Flag change							
Operand	Bit instruction							U	I	O	B	S	Z	D	C
	Absolute	Register direct	Address register indirect	Address register relative	SB relative	FB relative	FLG direct								
src	O	O	O	O	O	O		-	-	-	-	-	-	-	O
dest	O	O	O	O	O	O		-	-	-	-	-	-	-	-
src	O	O	O	O	O	O		-	-	-	-	-	-	-	-
dest	O	O	O	O	O	O	O	-	-	-	-	-	-	-	O ^f
src	O	O	O	O	O	O		-	-	-	-	-	-	-	O
src	O	O	O	O	O	O		-	-	-	-	-	-	-	O
dest	O	O	O	O	O	O		-	-	-	-	-	-	-	-
src	O	O	O	O	O	O		-	-	-	-	-	O	-	O
src	O	O	O	O	O	O		-	-	-	-	-	-	-	O
src	O	O	O	O	O	O		-	-	-	-	-	-	-	O
dest	O	O	O	O	O	O		-	-	-	-	-	-	-	-
src	O	O	O	O	O	O		-	-	-	-	-	O	-	O
dest	O	O	O	O	O	O		-	-	-	-	-	O	-	O
dest	O	O	O	O	O	O		-	-	-	-	-	O	-	O
src	O	O	O	O	O	O		-	-	-	-	-	-	-	O

*f Flag changes when C flag is specified for dest.

Arithmetic

Mnemonic	Explanation
ABS.size ^(Note) dest	$\text{dest} \leftarrow \text{dest} $;Absolute value of dest.
ADC.size src,dest	$\text{dest} \leftarrow \text{src} + \text{dest} + \text{C flag}$;Adds hexadecimal with carry.
ADCF.size dest	$\text{dest} \leftarrow \text{dest} + \text{C flag}$;Adds carry flag.
ADD.size src,dest	$\text{dest} \leftarrow \text{src} + \text{dest}$;Adds hexadecimal without carry.
ADDX src,dest	$\text{dest} \leftarrow \text{dest} - 32\text{-bit sign extension (src)}$;Hexadecimal addition without sign extension carry.
AND.size src,dest	$\text{dest} \leftarrow \text{src} \wedge \text{dest}$;Logical AND
CLIP.size src1,src2,dest	if $\text{src1} > \text{dest}$ then $\text{dest} \leftarrow \text{src1}$, if $\text{src2} > \text{dest}$ then $\text{dest} \leftarrow \text{src2}$;Clip instruction
CMP.size src,dest	$\text{dest} - \text{src}$;Comparison, with result determined by flag.
CMPX src,dest	$\text{dest} - 32\text{-bit sign extension (src)}$;Comparison, with result determined by flag.
DADC.size src,dest	$\text{dest} \leftarrow \text{src} + \text{dest} + \text{C flag}$;Decimal addition with carry.
DADD.size src,dest	$\text{dest} \leftarrow \text{src} + \text{dest}$;Decimal addition without carry.
DEC.size dest	$\text{dest} \leftarrow \text{dest} - 1$;Decrement
DIV.size src	$\text{R0 (Quotient), R2 (Remainder)} \leftarrow \text{R2R0} \div \text{src}$;Division with sign included.
DIVU.size src	$\text{R0 (Quotient), R2 (Remainder)} \leftarrow \text{R2R0} \div \text{src}$;Division without sign included.
DIVX.size src	$\text{R0 (Quotient), R2 (Remainder)} \leftarrow \text{R2R0} \div \text{src}$;Division with sign included.
DSBB.size src,dest	$\text{dest} \leftarrow \text{dest} - \text{src} - \text{C flag}$; Decimal subtraction with borrow.

Note:Write .W or .B for .size.

Operand	Addressing										Flag change							
	General instruction									Special instruction								
	Immediate	Absolute	Register direct	Address register indirect	Address register relative	SB relative	FB relative	Stack pointer relative	Control register direct	Program counter relative	U	I	O	B	S	Z	D	C
dest ^j		O	O	O	O	O	O				-	-	O	-	O	O	-	O
src	O	O	O	O	O	O	O				-	-	O	-	O	O	-	O
dest		O	O	O	O	O	O				-	-	O	-	O	O	-	O
dest ^j		O	O	O	O	O	O				-	-	O	-	O	O	-	O
src ^j	O	O	O	O	O	O	O				-	-	O	-	O	O	-	O
dest ^j		O	O	O	O	O	O		SP		-	-	O	-	O	O	-	O
src ^j	O	O	O	O	O	O	O				-	-	O	-	O	O	-	O
dest ^j		O	O	O	O	O	O				-	-	-	-	O	O	-	-
src ^j	O	O	O	O	O	O	O				-	-	-	-	O	O	-	-
dest ^j		O	O	O	O	O	O				-	-	-	-	O	O	-	-
src	O										-	-	O	-	O	O	-	O
dest ^j		O	O	O	O	O	O				-	-	O	-	O	O	-	O
src	O	O	O	O	O	O	O				-	-	-	-	O	O	-	O
dest		O	O	O	O	O	O				-	-	-	-	O	O	-	O
src	O	O	O	O	O	O	O				-	-	-	-	O	O	-	O
dest		O	O	O	O	O	O				-	-	-	-	O	O	-	O
dest ^j		O	O	O	O	O	O				-	-	-	-	O	O	-	-
src ^j	O	O	O	O	O	O	O				-	-	O	-	-	-	-	-
src ^j	O	O	O	O	O	O	O				-	-	O	-	-	-	-	-
src ^j	O	O	O	O	O	O	O				-	-	O	-	-	-	-	-
src	O	O	O	O	O	O	O				-	-	-	-	O	O	-	O
dest		O	O	O	O	O	O				-	-	-	-	O	O	-	O

*j Indirect addressing [dest] can be used in any register other than R0L/R0/R2R0, R0H/R2/-, R1H/R3/-, SP/SP/SP, dsp:8[SP], and #IMM.

Mnemonic	Explanation
DSUB.size ^(Note) src,dest	$\text{dest} \leftarrow \text{dest} - \text{src}$; Decimal subtraction without borrow.
INC.size dest	$\text{dest} \leftarrow \text{dest} + 1$;Increment.
MAX.size src,dest	if (src > dest) then $\text{dest} \leftarrow \text{src}$;Selects maximum value.
MIN.size src,dest	if (src < dest) then $\text{dest} \leftarrow \text{src}$;Selects minimum value.
MUL.size src,dest	$\text{dest} \leftarrow \text{dest} \times \text{src}$;Multiplication with sign included.
MULEX src	$\text{R1R2R0} \leftarrow \text{R2R0} \times \text{src}$;Extended multiplication with sign included .
MULU.size src,dest	$\text{dest} \leftarrow \text{dest} \times \text{src}$; Multiplication without sign included.
NEG.size dest	$\text{dest} \leftarrow 0 - \text{dest}$;Two's complement.
NOT.size dest	$\text{dest} \leftarrow \text{dest}$;Invert all bits.
OR.size src,dest	$\text{dest} \leftarrow \text{dest} \vee \text{src}$;Logical OR.
RMPA.size	Calculates sum-of-products using A0 as multiplicand address, A1 as multiplier address, and R3 as count.
ROLC.size dest	Rotates dest including C flag left by 1 bit ;Rotate left with carry.
RORC.size dest	Rotate dest including C flag right by 1 bit ;Rotate right with carry.
ROT.size src,dest	Rotate dest as many bits as indicated by signed src ;Rotate.
SBB.size src,dest	$\text{dest} \leftarrow \text{dest} - \text{src} - \text{C}$;Subtraction with borrow.

Note:Write .W or .B for .size.

Addressing											Flag change							
Operand	General instruction								Special instruction		U	I	O	B	S	Z	D	C
	Immediate	Absolute	Register direct	Address register indirect	Address register relative	SB relative	FB relative	Stack pointer relative	Control register direct	Program counter relative								
src	O	O	O	O	O	O	O				-	-	-	-	O	O	-	O
dest		O	O	O	O	O	O				-	-	-	-	O	O	-	O
dest ^{*k}		O	O	O	O	O	O				-	-	-	-	O	O	-	O
src	O	O	O	O	O	O	O				-	-	-	-	-	-	-	-
dest		O	O	O	O	O	O				-	-	-	-	-	-	-	-
src	O	O	O	O	O	O	O				-	-	-	-	-	-	-	-
dest		O	O	O	O	O	O				-	-	-	-	-	-	-	-
src ^{*k}	O	O	O	O	O	O	O				-	-	-	-	-	-	-	-
dest ^{*k}		O	O	O	O	O	O				-	-	-	-	-	-	-	-
src ^{*k}		O	O	O	O	O	O				-	-	-	-	-	-	-	-
src ^{*k}	O	O	O	O	O	O	O				-	-	-	-	-	-	-	-
dest ^{*k}		O	O	O	O	O	O				-	-	-	-	-	-	-	-
dest ^{*k}		O	O	O	O	O	O				-	-	O	-	O	O	-	O
dest ^{*k}		O	O	O	O	O	O				-	-	-	-	O	O	-	O
src ^{*k}	O	O	O	O	O	O	O				-	-	-	-	O	O	-	-
dest ^{*k}		O	O	O	O	O	O				-	-	-	-	O	O	-	-
											-	-	O	-	-	-	-	-
dest ^{*k}		O	O	O	O	O	O				-	-	-	-	O	O	-	O
dest ^{*k}		O	O	O	O	O	O				-	-	-	-	O	O	-	O
src ^{*k}	O ^{*l}	O ^{*m}									-	-	-	-	O	O	-	O
dest ^{*k}		O	O	O	O	O	O				-	-	-	-	O	O	-	O
src	O	O	O	O	O	O	O				-	-	O	-	O	O	-	O
dest		O	O	O	O	O	O				-	-	O	-	O	O	-	O

*k Indirect addressing [dest] can be used in any register other than R0L/R0/R2R0, R0H/R2/-, R1H/R3/-, SP/SP/SP, dsp:8[SP], and #IMM.

*l The range of possible values is $-8 < \#IMM4 < +8$.

*m Choose R1H.

Mnemonic	Explanation
<i>SCGEU/C</i> dest <i>SCLTU/NC</i> dest <i>SCEQ/Z</i> dest <i>SCNE/NZ</i> dest <i>SCGTU</i> dest <i>SCLEU</i> dest <i>SCPZ</i> dest <i>SCN</i> dest <i>SCGE</i> dest <i>SCLE</i> dest <i>SCGT</i> dest <i>SCLT</i> dest <i>SCO</i> dest <i>SCNO</i> dest	If C=1, dest \leftarrow 1;otherwise, dest \leftarrow 0 ;Set conditions. If C=0, dest \leftarrow 1;otherwise, dest \leftarrow 0 If Z=1, dest \leftarrow 1;otherwise, dest \leftarrow 0 If Z=0, dest \leftarrow 1;otherwise, dest \leftarrow 0 If C \wedge Z=1, dest \leftarrow 1;otherwise, dest \leftarrow 0 If C \wedge Z=0, dest \leftarrow 1;otherwise, dest \leftarrow 0 If S=0, dest \leftarrow 1;otherwise, dest \leftarrow 0 If S=1, dest \leftarrow 1;otherwise, dest \leftarrow 0 If S \vee O=0, dest \leftarrow 1;otherwise, dest \leftarrow 0 If (S \vee O) \vee Z=1, dest \leftarrow 1;otherwise, dest \leftarrow 0 If (S \vee O) \vee Z=0, dest \leftarrow 1;otherwise, dest \leftarrow 0 If S \vee O=1, dest \leftarrow 1;otherwise, dest \leftarrow 0 If O=1, dest \leftarrow 1;otherwise, dest \leftarrow 0 If O=0, dest \leftarrow 1;otherwise, dest \leftarrow 0
SCMPU.size	Compares strings successively in address incrementing direction using A0 as source address to compare and A1 as destination address to compare until comparison results in unmatch or source address becomes 0.
SHA.size src,dest	Arithmetically shifts dest as many bits as indicated by src. Bit overflowing from LSB (MSB) is transferred to C flag.
SHL.size src,dest	Logically shifts dest as many bits as indicated by src. Bit overflowing from LSB (MSB) is transferred to C flag.
SUB.size src,dest	dest \leftarrow dest - src ;Subtraction without borrow.
SUBX src,dest	dest \leftarrow dest - 32-bit sign extension (src) ;Extended subtraction without borrow.
TST.size src,dest	dest \wedge src ;Test
XOR.size ^(Note) src,dest	dest \leftarrow dest \vee src ;Exclusive logical OR.

Note:Write .W or .B for .size.

Addressing											Flag change							
Operand	General Instruction									Special instruction								
	Immediate	Absolute	Register direct	Address register indirect	Address register relative	SB relative	FB relative	Stack pointer relative	Control register direct	Program counter relative	U	I	O	B	S	Z	D	C
dest ⁿ		O	O	O	O	O	O				-	-	-	-	-	-	-	-
src											-	-	O	-	O	O	-	O
src ⁿ	O ^o		R1H								-	-	O	-	O	O	-	O
dest ⁿ		O	O	O	O	O	O				-	-	O	-	O	O	-	O
src ⁿ	O ^o		R1H								-	-	O	-	O	O	-	O
dest ⁿ		O	O	O	O	O	O				-	-	O	-	O	O	-	O
src ⁿ	O	O	O	O	O	O	O				-	-	O	-	O	O	-	O
dest ⁿ		O	O	O	O	O	O				-	-	O	-	O	O	-	O
src	O	O	O	O	O	O	O				-	-	-	-	O	O	-	-
dest		O	O	O	O	O	O				-	-	-	-	O	O	-	-
src ⁿ	O	O	O	O	O	O	O				-	-	-	-	O	O	-	-
dest ⁿ		O	O	O	O	O	O				-	-	-	-	O	O	-	-

*n Indirect addressing [src] and [dest] can be used in any register other than R0L/R0/R2R0, R0H/R2/-, R1H/R3/-, SP/SP/SP, dsp:8[SP], and #IMM.

*o When (.size) is (.B) or (.W), the range of possible values is $-8 < \#IMM4 < +8$ ((0)); when (.L), the range of possible values is $-16 < \#IMM8 < +16$ ((0)).

JUMP

Mnemonic		Explanation
ADJNZ.size ^(Note)	src,dest,label	dest ← dest + src If result of dest + src is not 0, jump to label. ;Add and conditional branch.
JGEU/C	label	If C=1, jump to label ;otherwise, execute next instruction. ;Conditional branch.
JLTU/NC	label	If C=0, jump to label ;otherwise, execute next instruction.
JEQ/Z	label	If Z=1, jump to label ;otherwise, execute next instruction.
JNE/NZ	label	If Z=0, jump to label ;otherwise, execute next instruction.
JGTU	label	If C ∧ Z=1, jump to label ;otherwise, execute next instruction.
JLEU	label	If C ∧ Z=0, jump to label ;otherwise, execute next instruction.
JPZ	label	If S=0, jump to label ;otherwise, execute next instruction.
JN	label	If S=1, jump to label ;otherwise, execute next instruction.
JGE	label	If S ∨ O=0, jump to label ;otherwise, execute next instruction.
JLE	label	If (S ∨ O) ∨ Z=1, jump to label ;otherwise, execute next instruction.
JGT	label	If (S ∨ O) ∨ Z=0, jump to label ;otherwise, execute next instruction.
JLT	label	If S ∨ O=1, jump to label ;otherwise, execute next instruction.
JO	label	If O=1, jump to label ;otherwise, execute next instruction.
JNO	label	If O=0, jump to label ;otherwise, execute next instruction.
JMP	label	Jump to label. ;Unconditional branch.
JMPI	src	Jump to address indicated by src. ;Indirect branch.
JMPS	src	Branches using special page vector table.
JSR	label	Subroutine call.
JSRI	src	Indirect subroutine call.
JSRS	src	Calls subroutine using special page vector table.
RTS		Return from subroutine.
SBJNZ.size	src,dest,label	Branches to label if the result of dest ← dest - src is not 0 ;Subtraction & conditional branch.

Note:Write .W or .B for .size.

Addressing										Flag change								
Operand	General instruction								Special instruction		U	I	O	B	S	Z	D	C
	Immediate	Absolute	Register direct	Address register indirect	Address register relative	SB relative	FB relative	Stack pointer relative	Control register direct	Program counter relative								
src	O ^p										-	-	-	-	-	-	-	
dest		O	O	O	O	O	O				-	-	-	-	-	-	-	
label										label ^q								
label										label ^r	-	-	-	-	-	-	-	
label		O ^s								label ^t	-	-	-	-	-	-	-	
src		O	O	O	O	O	O				-	-	-	-	-	-	-	
src		O	O	O	O	O	O				-	-	-	-	-	-	-	
src	O ^u										-	-	-	-	-	-	-	
src		O	O	O	O	O	O				-	-	-	-	-	-	-	
src	O ^u										-	-	-	-	-	-	-	
											-	-	-	-	-	-	-	
src	O ^v										-	-	-	-	-	-	-	
dest		O	O	O	O	O	O				-	-	-	-	-	-	-	
label										label ^q								

*p The range of immediate values is $-8 < \#IMM4 < +7$.

*q The range of label is $PC - 126 < label < PC + 129$. PC is the start address of the instruction.

*r The range of label is $PC - 127 < label < PC + 128$. PC is the start address of the instruction.

*s This is a 24-bit absolute address.

*t The range of label is $PC - 32767 < label < PC + 32768$. PC is the start address of the instruction.

*u #IMM8 is a special page address.

*v The range of immediate values is $-7 < \#IMM4 < +8$.

Sign extension

Mnemonic	Explanation
EXTS.size ^(Note) dest	dest ← Sign extension (dest) conforming to .size.
EXTS.size src,dest	dest ← Sign extension (src) conforming to .size.
EXTZ src,dest	dest ← Zero extension(src) of 16 bits.

Index

Mnemonic	Explanation
INDEXB.size ^(Note) src	Modifies next instruction addressing in units of bytes.
INDEXBD.size src	
INDEXBS.size src	
INDEXW.size src	Modifies next instruction addressing in units of words.
INDEXWD.size src	
INDEXWS.size src	
INDEXL.size src	Modifies next instruction addressing in units of long words.
INDEXLD.size src	
INDEXLS.size src	

Note: Write .W or .B for .size.

Addressing											Flag change							
Operand	General instruction								Special instruction		U	I	O	B	S	Z	D	C
	Immediate	Absolute	Register direct	Address register indirect	Address register relative	SB relative	FB relative	Stack pointer relative	Control register direct	Program counter relative								
dest		O	O	O	O	O	O				-	-	-	-	O	O	-	-
src		O	O		O	O	O				-	-	-	-	O	O	-	-
dest		O	O	O	O	O	O				-	-	-	-	O	O	-	-
src		O	O		O	O	O				-	-	-	-	O	O	-	-
dest		O	O	O	O	O	O				-	-	-	-	O	O	-	-

Addressing										Flag change							
Operand	General instruction								Special instruction	U	I	O	B	S	Z	D	C
	Immediate	Absolute	Register direct	Address register indirect	Address register relative	SB relative	FB relative	Stack pointer relative	Control register direct	Program counter relative							
src		O	O	O	O	O	O				-	-	-	-	-	-	-
src		O	O	O	O	O	O				-	-	-	-	-	-	-
src		O	O	O	O	O	O				-	-	-	-	-	-	-

High-level language and OS support

Mnemonic		Explanation
ENTER	src	Generates stack frame.
EXITD	src	Frees stack frame.
LDCTX	abs16,abs24	Restores context.
STCTX	abs16,abs24	Saves context.

Addressing											Flag change							
Operand	General instruction									Special instruction	U	I	O	B	S	Z	D	C
	Immediate	Absolute	Register direct	Address register indirect	Address register relative	SB relative	FB relative	Stack pointer relative	Control register direct	Program counter relative								
src	O										-	-	-	-	-	-	-	-
src		O	O	O	O	O	O				-	-	-	-	-	-	-	-
abs16 ^w											-	-	-	-	-	-	-	-
abs24 ^w											-	-	-	-	-	-	-	-
abs16 ^w											-	-	-	-	-	-	-	-
abs24 ^w											-	-	-	-	-	-	-	-

*w For abs16, set the RAM address in which the task number is stored; for abs24, set the start address of table data.

Other

Mnemonic	Explanation
BRK	Generate BRK interrupt.
BRK2	Debugger-only interrupt, whose use in user program is therefore inhibited.
FCLR dest	Clears flags in flag register to 0.
FREIT	Returns from interrupt routine after fast interrupt request.
FSET	Sets flags in flag register to 1.
INT	Generates software interrupt.
INTO	Generates overflow interrupt when O (overflow) flag = 1.
LDC src,dest	Transfers src to dedicated register indicated by dest.
LDIPL src	Transfers src to IPL.
NOP	No operation.
REIT	Returns from interrupt routine.
STC src,dest	Transfers from dedicated register indicated by src to dest.
UND	Generates undefined instruction interrupt.
WAIT	Stops executing program.

Addressing											Flag change							
Operand	General instruction									Special instruction								
	Immediate	Absolute	Register direct	Address register indirect	Address register relative	SB relative	FB relative	Stack pointer relative	Control register direct	Program counter relative	U	I	O	B	S	Z	D	C
											O	O	-	-	-	-	O	-
											O	O	-	-	-	-	O	-
dest									O		Selected flag is cleared to 0.							
											Becomes the content of SVF.							
dest											Selected flag is set to 1.							
src	O ^x										O	O	-	-	-	-	O	-
											O	O	-	-	-	-	O	-
src	O	O	O	O	O	O	O				Flag changes only when dest is FLG.							
dest									O ^y									
src	O ^z										-	-	-	-	-	-	-	-
											-	-	-	-	-	-	-	-
											Returns to FLG state before interrupt request was accepted.							
src	O ^y										-	-	-	-	-	-	-	-
dest		O	O	O	O	O	O				-	-	-	-	-	-	-	-
											O	O	-	-	-	-	O	-
											-	-	-	-	-	-	-	-

*x #IMM6 specifies a software interrupt number.

*y Any dedicated register except the PC register can be selected.

*z The range of possible values is $0 < \text{\#IMM3} < 7$.

2.6.3 Transfer Instructions

Transfers normally are performed in bytes or words. There are 14 transfer instructions available. Included among these are a 4-bit transfer instruction that transfers only 4 bits, a conditional store instruction that is combined with conditional branch, and a string instruction that transfers data collectively.

This section explains these three characteristic instructions of the M16C/80 series among its data transfer-related instructions.

4 Bit Transfer Instruction

This instruction transfers 4 high-order or low-order bits of an 8-bit register or memory. This instruction can be used for generating unpacked BCD code or I/O port input/output in 4 bits. The mnemonic placed in *Dir* varies depending on whether the instruction is used to transfer high-order or low-order 4 bits. When using this instruction, be sure to use R0L for src or dest.

Table 2.6.5 4 Bit Transfer Instruction

Mnemonic	Description Format	Explanation
MOV <i>Dir</i>	MOVHH src,dest4	Transfer high-order bits: src → 4 high-order bits: dest
	MOVHL src,dest4	high-order bits: src → 4 low-order bits: dest
	MOVLH src,dest4	low-order bits: src → 4 high-order bits: dest
	MOVLL src,dest4	low-order bits: src → 4 low-order bits: dest

Note: Either src or dest must always be R0L.

Conditional Store Instruction

This is a conditional transfer instruction that uses the Z flag state as the condition of transfer. This instruction allows the user to perform condition determination and data transfer in one instruction. There are three types of conditional store instructions: STZ, STNZ, and STZX. Figure 2.6.3 shows an example of how the instruction works.

Table 2.6.6 Conditional Store Instruction

Mnemonic	Description Format	Explanation
STZ	STZ src,dest	Transfers src to dest when Z flag = 1.
STNZ	STNZ src,dest	Transfers src to dest when Z flag = 0.
STZX	STZX src1,src2,dest	Transfers src1 to dest when Z flag = 1. Transfers src2 to dest when Z flag = 0.

Note: Only #IMM8/16 (8/16-bit immediate) can be used for src, src1, and src2.

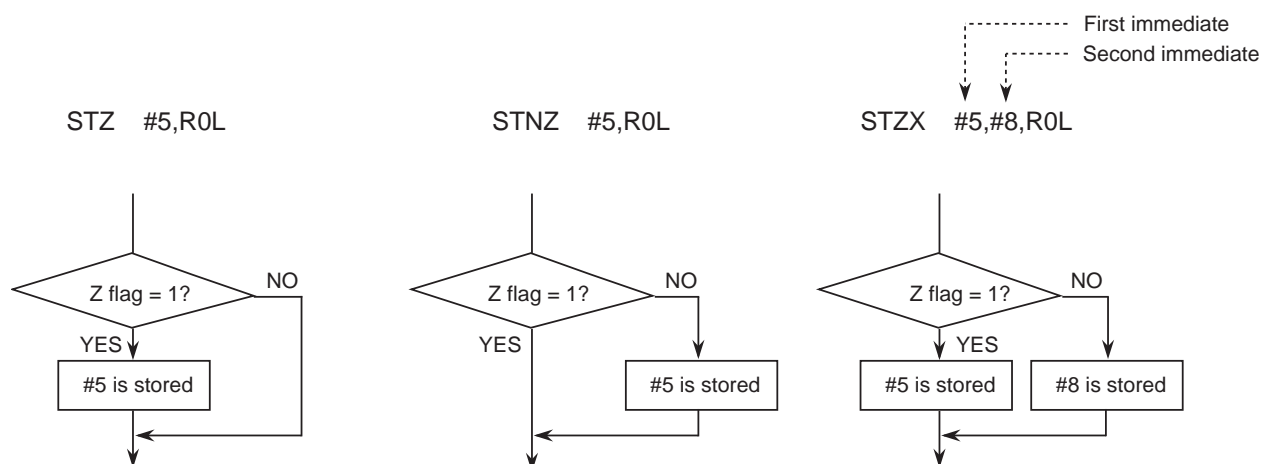


Figure 2.6.3 Typical operations of conditional store instructions

String Instruction

This instruction transfers data collectively. Use it for transferring blocks and clearing RAM. Set the source address, destination address, and transfer count in each register before executing the instruction, as shown in Figure 2.6.4. Data is transferred in bytes or words. Figure 2.6.5 shows an example of how the string instruction works.

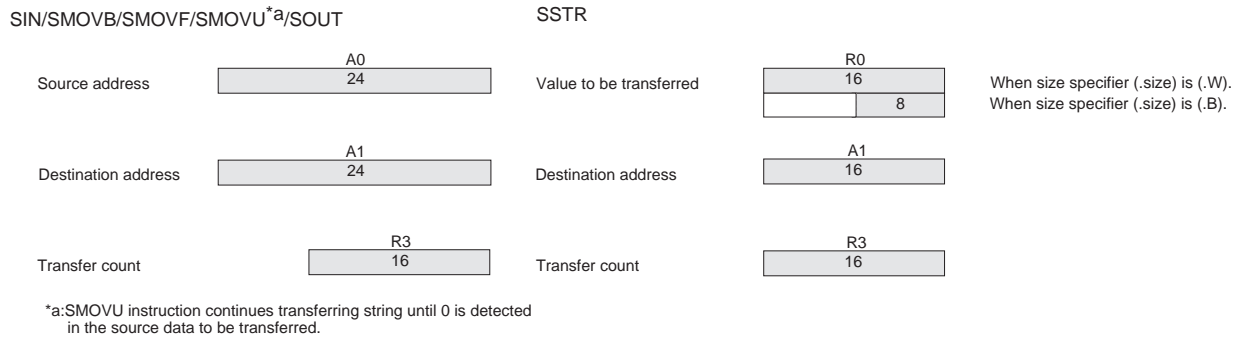


Figure 2.6.4 Setting registers for string instructions

Table 2.6.7 String Instruction

Mnemonic	Description Format	Explanation
SIN	SIN.B SIN.W	Transfers string in address incrementing direction (source of transfer fixed).
SOUT	SOUT.B SOUT.W	Transfers string in address incrementing direction (destination of transfer fixed).
SMOVB	SMOVB.B SMOVB.W	Transfers string in address decrementing direction.
SMOVF	SMOVF.B SMOVF.W	Transfers string in address incrementing direction.
SMOVU	SMOVU.B SMOVU.W	Continues transferring string in address incrementing direction until 0 is detected.
SSTR	SSTR.B SSTR.W	Stores string in address incrementing direction.

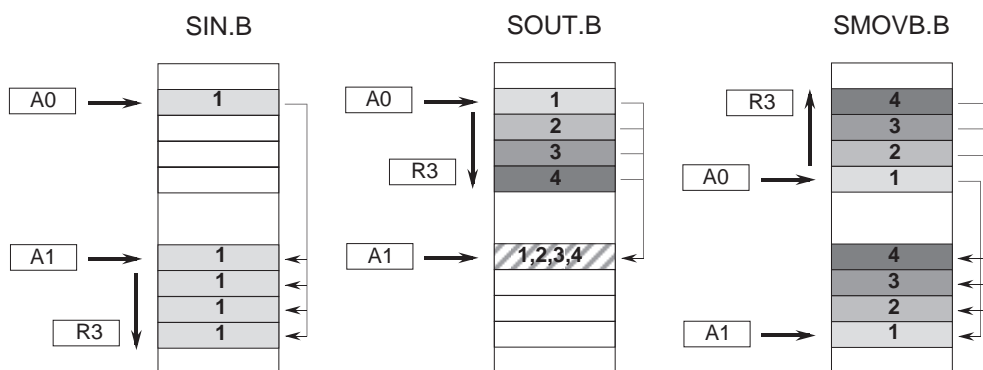


Figure 2.6.5 Typical operations of string instructions 1

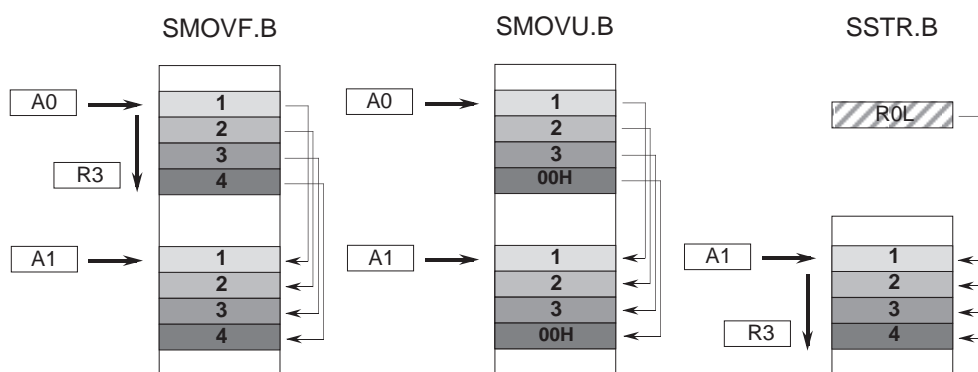


Figure 2.6.6 Typical operations of string instructions 2

2.6.4 Arithmetic Instructions

There are 39 arithmetic instructions available. This section explains the characteristic arithmetic instructions of the M16C/80 series.

Multiply Instruction

Multiply instructions are classified into signed multiply instructions, extended signed multiply instructions, and unsigned multiply instructions. The signed multiply instructions and unsigned multiply instructions allow the size to be specified. When .B is specified, operation is performed in 8 bits, i.e., (8 bits) x (8 bits) = (16 bits). When .W is specified, operation is performed in 16 bits, i.e., (16 bits) x (16 bits) = (32 bits).

When .B is specified, address registers cannot be used for both src and dest. Note also that the flag does not change in multiply instructions. An example of how multiply instructions operate is shown in Figure 2.6.7.

Table 2.6.8 Multiply Instruction

Mnemonic	Description Format	Explanation
MUL	MUL.B src,dest MUL.W src,dest	Signed multiply instruction $\text{dest} \leftarrow \text{desc} \times \text{src}$
MULEX	MULEX src,dest MULEX src,dest	Extended signed multiply instruction $\text{R1R2R0} \leftarrow \text{R2R0} \times \text{src}$
MULU	MULU.B src,dest MULU.W src,dest	Unsigned multiply instruction $\text{dest} \leftarrow \text{dest} \times \text{src}$

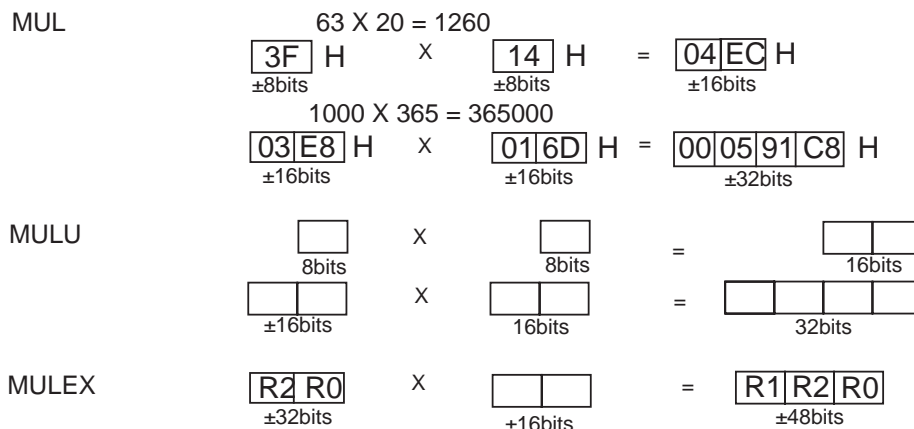


Figure 2.6.7 Typical operations of multiply instructions

Divide Instruction

There are three types of divide instructions: two signed divide instructions and one unsigned divide instruction. All these three instructions allow the user to specify the desired size. When .B is specified, calculation is performed in (16 bits) ÷ (8 bits) = (8 bits)... (Remainder in 8 bits); when .W is specified, calculation is performed in (32 bits) ÷ (16 bits) = (16 bits)... (Remainder in 16 bits). In divide instructions, the O flag changes state when the result overflows or a divide by 0 is attempted. An example of how divide instructions operate is shown in Figure 2.6.8.

Table 2.6.9 Divide Instruction

Mnemonic	Description Format	Explanation
DIV	DIV.B src,dest DIV.W src,dest	Signed divide instruction Sign of remainder matches that of dividend.
DIVX	DIVX.B src,dest DIVX.W src,dest	Signed divide instruction Sign of remainder matches that of divisor.
DIVU	DIVU.B src,dest DIVU.W src,dest	Unsigned divide instruction.

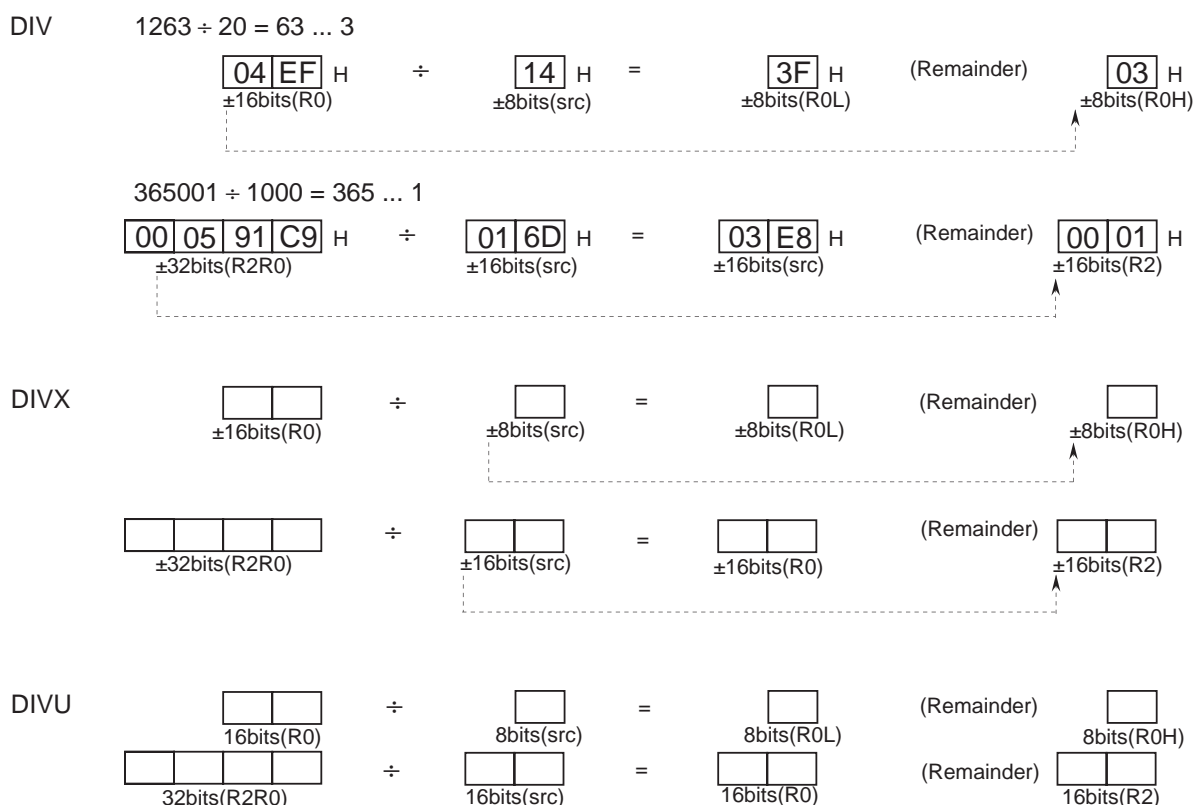


Figure 2.6.8 Typical operations of divide instructions

Difference between DIV and DIVX Instructions

Both DIV and DIVX are signed divide instructions. The difference between these two instructions is the sign of the remainder.

As shown in Table 2.6.10, the sign of the remainder deriving from the DIV instruction is the same as that of the dividend. With the DIVX instruction, however, the sign is the same as that of the divisor.

Table 2.6.10 Difference between DIV and DIVX Instructions

DIV	$33 \div 4 = 8 \dots 1$	The sign of the remainder is the same as that of the dividend.
	$33 \div (-4) = 8 \dots 1$	
	$-33 \div 4 = 8 \dots (-1)$	
DIVX	$33 \div 4 = 8 \dots 1$	The sign of the remainder is the same as that of the divisor.
	$33 \div (-4) = 9 \dots (-3)$	
	$-33 \div 4 = -9 \dots 3$	

Decimal Add Instruction

There are two types of decimal add instructions: one with a carry and the other without a carry. The S, Z, and C flags change state when the decimal add instruction is executed. Figure 2.6.9 shows an example of how these instructions operate.

Table 2.6.11 Decimal Add Instruction

Mnemonic	Description Format	Explanation
DADD	DADD.B src,dest DADD.W src,dest	Add in decimal not including carry.
DADC	DADC.B src,dest DADC.W src,dest	Add in decimal including carry.

DADD 2digits 62 + 50 = 112

	10's place	1's place
	6	2
+	5	0
	1	1 2
C flag	1	

4digits 1234 + 9000 = 10234

	1000's place	100's place	10's place	1's place
	1	2	3	4
+	9	0	0	0
	1	0	2	3 4
C flag	1			

DADC 2digits 62 + 30 + C flag 1 = 93

	10's place	1's place
	6	2
	3	0
+		C flag 1
	0	9 3
C flag	0	

4digits 1234 + 9000 + C flag 1 = 10234

	1000's place	100's place	10's place	1's place
	1	2	3	4
	9	0	0	0
+				C flag 1
	1	0	2	3 5
C flag	1			

Figure 2.6.9 Typical operations of decimal add instructions

Decimal Subtract Instruction

There are two types of decimal subtract instructions: one with a borrow and the other without a borrow.
The S, Z, and C flags change state when the decimal subtract instruction is executed. Figure 2.6.10 shows an example of how these instructions operate.

Table 2.6.12 Decimal Subtract Instruction

Mnemonic	Description Format	Explanation
DSUB	DSUB.B src,dest DSUB.W src,dest	Subtract in decimal not including borrow.
DSBB	DSBB.B src,dest DSBB.W src,dest	Subtract in decimal including borrow.

DSUB 2digits 78 - 11 = 67

	10's place	1's place
	7	8
-	1	1
	1	6
	6	7
C flag	1	

4digits 1111 - 1234 = 9877

	1000's place	100's place	10's place	1's place
	1	1	1	1
-	1	2	3	4
	0	9	8	7
	9	8	7	7
C flag	0			

DSBB 2digits 78 - 11 - C flag 1 = 67

	10's place	1's place
	7	8
	1	1
-		Cflag 0
	1	6
	6	7
C flag	1	

4digits 1234 - 1111 - C flag 0 = 0122

	1000's place	100's place	10's place	1's place
	1	1	1	1
	1	2	3	4
-				Cflag 1
	0	9	8	7
	0	9	8	7
C flag	0			

Figure 2.6.10 Typical operations of decimal subtract instructions

Sum of Products Calculate Instruction

This instruction calculates a sum of products and if an overflow occurs during calculation, generates an overflow interrupt. Set the multiplicand address, multiplier address, and sum of products calculation count in each register as shown in Figure 2.6.11. Figure 2.6.12 shows an example of how the sum-of-products calculate instruction works.

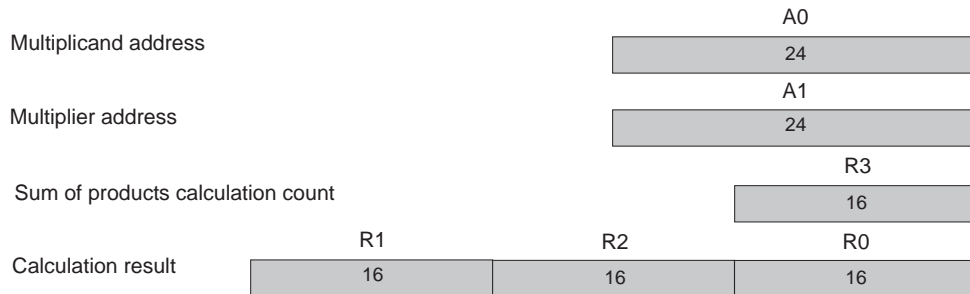


Figure 2.6.11 Setting registers for sum-of-products calculation instruction

Table 2.6.13 Sum of Products Calculate Instruction

Mnemonic	Description Format	Explanation
RMPA	RMPA.B RMPA.W	Calculates a sum of products using A0 as multiplicand address, A1 as multiplier address, and R3 as operation count.

Note1: If an overflow occurs during calculation, the overflow flag (O flag) is set to 1 before terminating the calculation.

Note2: If an interrupt is requested during calculation, the sum of products calculation count is subtracted after completing the addition in progress before accepting the interrupt request..

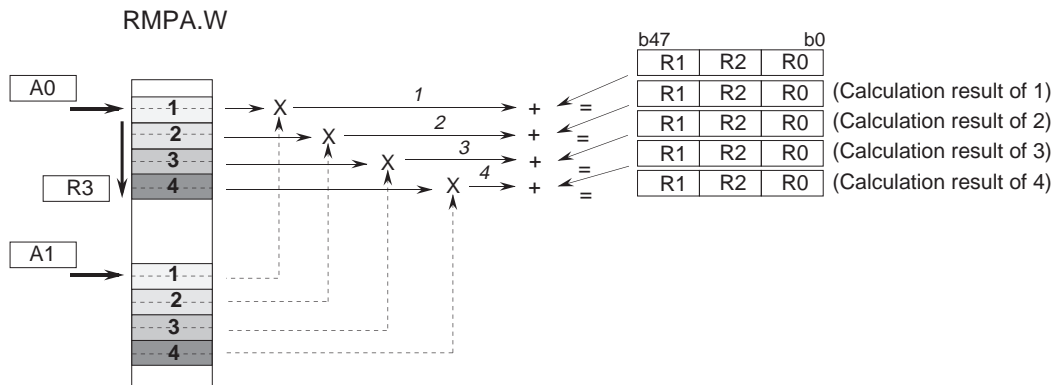


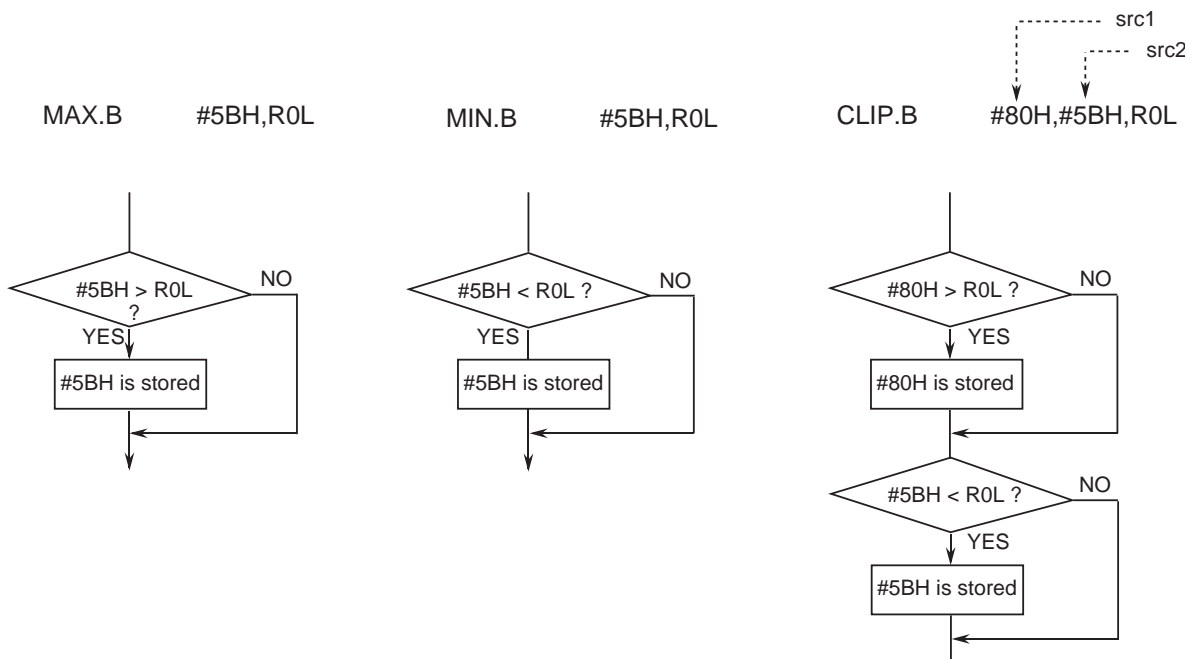
Figure 2.6.12 Typical operation of sum-of-products calculation instruction

MAX, MIX, and CLIP instructions

The M16C/80 has three instructions that allow the lower-limit and upper-limit values of data to be set in one instruction. The CLIP instruction is a combination of MAX and MIN instructions. In these instructions, the flag does not change. An example of how these instructions operate is shown in Figure 2.6.13.

Table 2.6.14 MAX, MIX, and CLIP instructions

Mnemonic	Description Format	Explanation
MAX	MAX.B src,dest MAX.W src,dest	Compares src and dest with sign included and transfers src to dest when src is greater than dest.
MIN	MIN.B src,dest MIN.W src,dest	Compares src and dest with sign included and transfers src to dest when src is smaller than dest.
CLIP	CLIP.B src1,src2,dest CLIP.W src1,src2,dest	Compares src and dest with sign included and transfers src1 to dest when src1 is greater than dest; next, transfers src2 to dest when src2 is smaller than dest. Consequently, nothing is stored if $\text{src1} \leq \text{dest} \leq \text{src2}$.

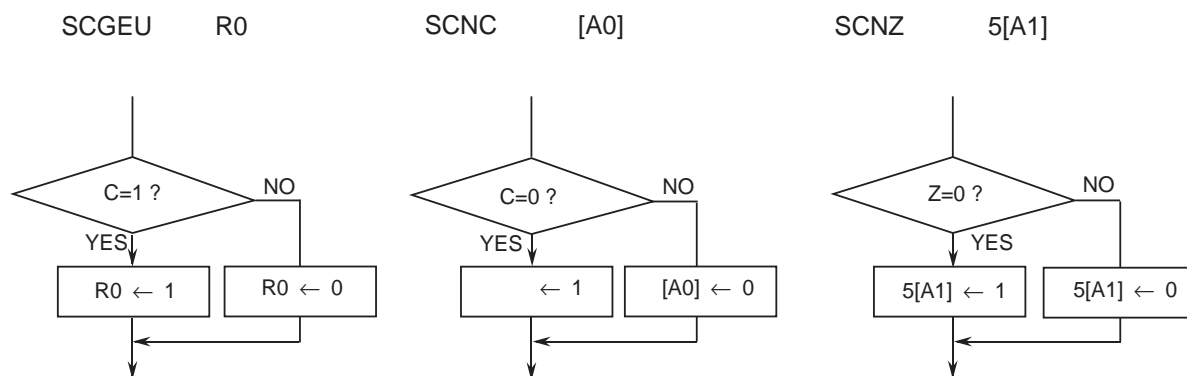
**Figure 2.6.13 Typical operation of MAX, MIX, and CLIP instructions**

SCcnd instruction

The M16C/80 has an instruction that stores a 1 or 0 in dest (1 word) depending on the flag content of the flag register. An example of how this instruction operates is shown in Figure 2.6.14.

Table 2.6.15 SCcnd instruction

Mnemonic	Description Format	Explanation
SCcnd dest	SCGEU/C dest	If C=1, dest \leftarrow 1; otherwise, dest \leftarrow 0
	SCLTU/NC dest	If C=0, dest \leftarrow 1; otherwise, dest \leftarrow 0
	SCEQ/Z dest	If Z=1, dest \leftarrow 1; otherwise, dest \leftarrow 0
	SCNE/NZ dest	If Z=0, dest \leftarrow 1; otherwise, dest \leftarrow 0
	SCGTU dest	If $C \wedge Z=1$, dest \leftarrow 1; otherwise, dest \leftarrow 0
	SCLEU dest	If $C \wedge Z=0$, dest \leftarrow 1; otherwise, dest \leftarrow 0
	SCPZ dest	If S=0, dest \leftarrow 1; otherwise, dest \leftarrow 0
	SCN dest	If S=1, dest \leftarrow 1; otherwise, dest \leftarrow 0
	SCGE dest	If $S \vee O=0$, dest \leftarrow 1; otherwise, dest \leftarrow 0
	SCLE dest	If $(S \vee O) \vee Z=1$, dest \leftarrow 1; otherwise, dest \leftarrow 0
	SCGT dest	If $(S \vee O) \vee Z=0$, dest \leftarrow 1; otherwise, dest \leftarrow 0
	SCLT dest	If $S \vee O=1$, dest \leftarrow 1; otherwise, dest \leftarrow 0
	SCO dest	If O=1, dest \leftarrow 1; otherwise, dest \leftarrow 0
	SCNO dest	If O=0, dest \leftarrow 1; otherwise, dest \leftarrow 0

**Figure 2.6.14 Typical operation of SCcnd instruction**

2.6.5 Branch Instructions

There are ten branch instructions available with the M16C/80 series. This section explains some characteristic branch instructions among these.

Unconditional Branch Instruction

This instruction causes control to jump to label unconditionally.

The jump distance specifier normally is omitted. When this specifier is omitted, the assembler optimizes the jump distance when assembling the program. Figure 2.6.16 shows an example of how the unconditional branch instruction works.

Table 2.6.16 Unconditional Branch Instruction

Mnemonic	Description Format	Explanation
JMP	JMP.S label	Jumps to label.
	JMP.B label	
	JMP.W label	
	JMP.A label	

Range of jump:

.S Jump in PC relative addressing from +2 to +9 (operand : 0 byte)

.B Jump in PC relative addressing from -127 to +128 (operand : 1 byte)

.W Jump in PC relative addressing from -32,767 to +32,768 (operand : 2 bytes)

.A Jump in 20 bit absolute addressing (operand : 3 bytes)

JMP LABEL1

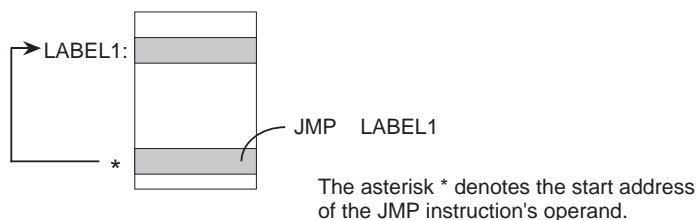


Figure 2.6.15 Typical operation of unconditional Branch Instruction

Indirect Branch Instruction

This instruction indirectly branches to the address indicated by src. When the branch distance specifier ".W" is specified, the program branches to the address derived by adding src to the start address of the JMP instruction, with the sign included. If src is memory, the necessary memory size is 2 bytes. When the branch distance specifier ".A" is specified, the program branches to the address indicated by src. If src is memory, the necessary memory size is 3 bytes. This instruction always requires that a branch distance specifier be specified. An example of how the indirect branch instruction operates is shown in Figure 2.6.16.

Table 2.6.17 Indirect Branch Instruction

Mnemonic	Description Format	Explanation
JMPI	JMPI.W src JMPI.A src	Jumps indirectly to the address indicated by src.

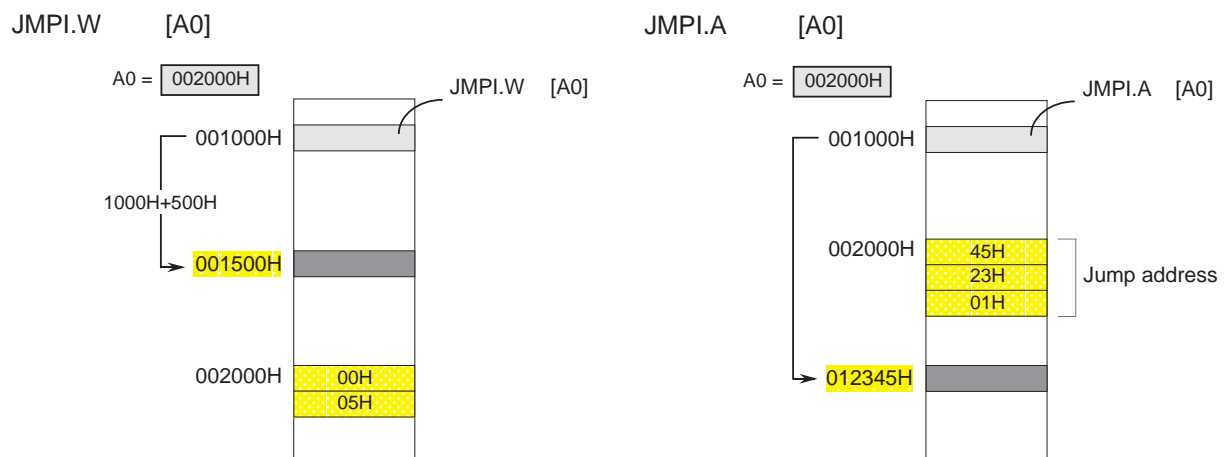


Figure 2.6.16 Typical operation of indirect branch instruction

Special Page Branch Instruction

This instruction branches to the address derived by adding FF0000H to the address that has been set in the relevant special page vector table. The address range in which branch occurs is FF0000H to FFFFFFFH. Because the special page branch instruction is only 2 bytes in size, it helps to increase ROM efficiency.

Use a special page number or label to specify the target address. Make sure the special page number is prefixed by "#," and that label is prefixed by "\". When label is used for address specification, the assembler calculates the special page number. An example of how the special page branch instruction operates is shown in Figure 2.6.17.

Table 2.6.18 Special Page Branch Instruction

Mnemonic	Description Format
JMPS	JMPS # special page number
JMPS	JMPS \ special page vector address 18 ≤ special page number ≤ 255

JMPS #251

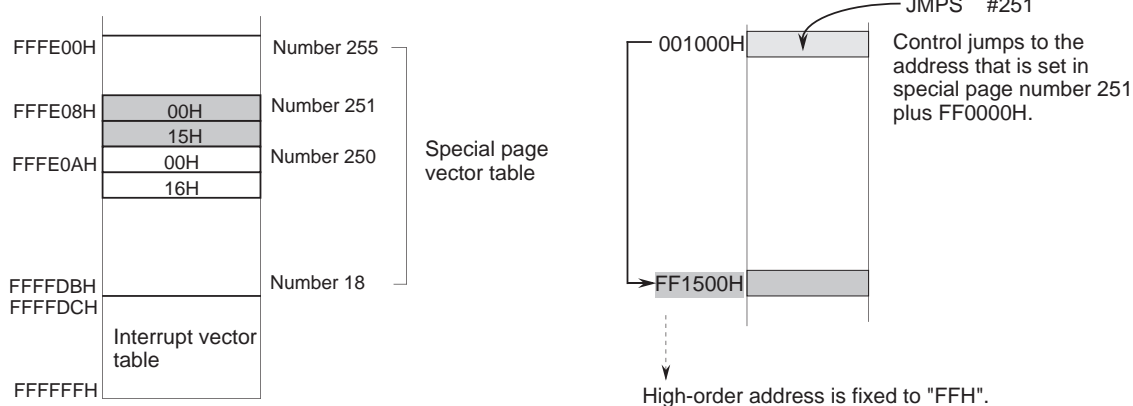


Figure 2.6.17 Typical operation of special page branch instruction

Conditional Branch Instruction

This instruction examines flag status with respect to the conditions listed below and causes control to branch if the condition is true or executes the next instruction if the condition is false. Figure 2.6.18 shows an example of how the conditional branch instruction works.

Table 2.6.19 Conditional Branch Instruction

Mnemonic	Description Format	Explanation
JCnd	JCnd label	Jumps to label if condition is true, or executes next instruction if condition is false.

Cnd	True/false determining conditions (14 conditions)	
GEU/C	$C = 1$	Equal or greater/ Carry flag = 1.
GTU	$C = 1 \ \& \ Z = 0$	Unsigned and greater.
EQ/Z	$Z = 1$	Equal/ Zero flag = 1.
N	$S = 1$	Negative.
LE	$(z = 1) \mid (S = 1 \ \& \ O = 0) \mid (S = 0 \ \& \ O = 1)$	Equal or signed and smaller.
O	$O = 1$	Overflow flag = 1.
GE	$(S = 1 \ \& \ O = 1) \mid (S = 0 \ \& \ O = 0)$	Equal or signed and greater.
LTU/NC	$C = 0$	Smaller/ Carry flag = 0.
LEU	$C = 0 \mid Z = 1$	Equal or smaller.
NE/NZ	$Z = 0$	Not equal/ Zero flag = 0.
PZ	$S = 0$	Positive or zero.
GT	$(S = 1 \ \& \ O = 1 \ \& \ Z = 0) \mid (S = 0 \ \& \ O = 0 \ \& \ Z = 0)$	Signed and greater.
NO	$O = 0$	Overflow flag = 0.
LT	$(S = 1 \ \& \ O = 0) \mid (S = 0 \ \& \ O = 1)$	Signed and smaller.

Range of jmp : -127 to +128

JEQ LABEL1

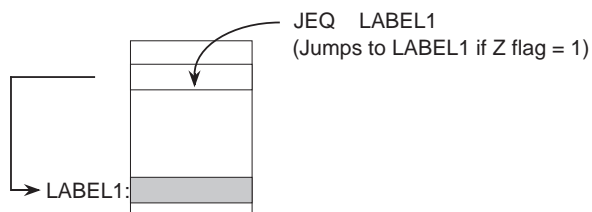


Figure 2.6.18 Typical operation of conditional branch instruction

Add (Subtract) & Conditional Branch Instruction

This instruction is convenient for determining whether repeat processing is terminated or not. The values added or subtracted by this instruction are limited to 4-bit immediate. Specifically, the value is -8 to +7 for the ADJNZ instruction, and -7 to +8 for the SBJNZ instruction. The range of addresses to which control can jump is -126 to +129 from the start address of the ADJNZ/SBJNZ instruction. Figure 2.6.8 shows an example of how the add (subtract) & conditional branch instruction works.

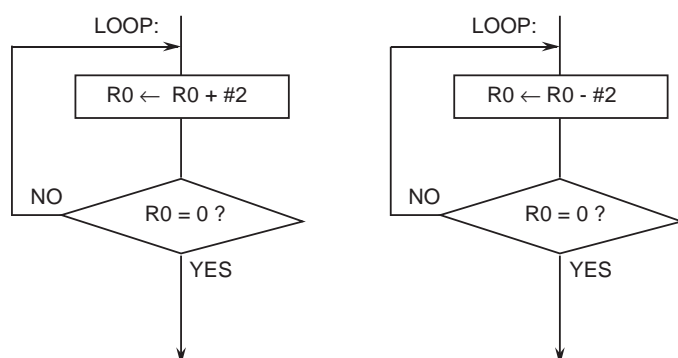
Table 2.6.20 Add (Subtract) & Conditional Branch Instruction

Mnemonic	Description Format	Explanation
ADJNZ	ADJNZ.B #IMM4,dest,label	Adds immediate to dest.
	ADJNZ.W #IMM4,dest,label	Jump to label if result is not 0.
SBJNZ	SBJNZ.B #IMM4,dest,label	Subtracts immediate from dest.
	SBJNZ.W #IMM4,dest,label	Jump to label if result is not 0.

Note1: #IMM can only be a 4 bit immediate (-8 to +7 for the ADJNZ instruction ; -7 to +8 for the SBJNZ instruction).

Note2: The range of addresses to which control can jump in PC relative addressing is -126 to +129

ADJNZ.W #2,R0,LOOP SBJNZ.W #2,R0,LOOP



from the start address of the ADJNZ / SBJNZ instructions.

Figure 2.6.19 Typical operations of add (subtract) & conditional branch instructions

2.6.6 Bit Instructions

This section explains the bit instructions of the M16C/80 series.

Logical Bit Manipulating Instruction

This instruction ANDs or ORs a specified register or memory bit and the C flag and stores the result in the C flag. Figure 2.6.12 shows an example of how the logical bit manipulating instruction works.

Table 2.6.21 Logical Bit Manipulating Instruction

Mnemonic	Description Format	Explanation
BAND	BAND src	$C\text{ flag} \leftarrow \text{src} \wedge C\text{ flag}$;Bitwise AND.
BNAND	BNAND src	$C\text{ flag} \leftarrow \text{src} \wedge C\text{ flag}$;Inverted bitwise AND.
BNOR	BNOR src	$C\text{ flag} \leftarrow \text{src} \vee C\text{ flag}$;Inverted bitwise OR.
BNXOR	BNXOR src	$C\text{ flag} \leftarrow \text{src} \vee C\text{ flag}$;Inverted bitwise exclusive OR.
BOR	BOR src	$C\text{ flag} \leftarrow \text{src} \vee C\text{ flag}$;Bitwise OR.
BXOR	BXOR src	$C\text{ flag} \leftarrow \text{src} \vee C\text{ flag}$;Bitwise exclusive OR.

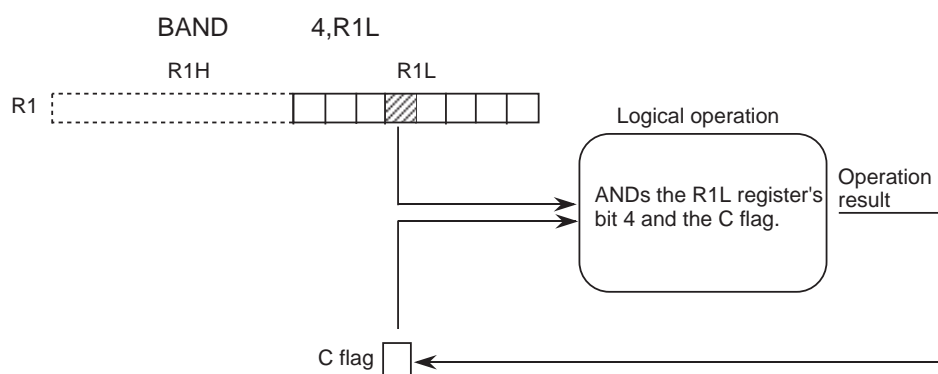


Figure 2.6.20 Typical operation of logical bit manipulating instruction

This instruction transfers a bit from depending on whether a condition is met. If the condition is true, it transfers "1"; if the condition is false, it transfers "0". In all cases, a flag is used to determine whether the condition is true or false. This instruction must be preceded by an instruction that causes the flag to change. Figure 2.6.21 shows an example of how the conditional bit transfer instruction works.

Mnemonic	Description Format	Explanation
BMCnd	BMCnd dest C	Transfers "1" if condition is true or "0" if condition is false.

Cnd	True/false determining conditions (14 conditions)	
GEU/C	$C = 1$	Equal or greater/ Carry flag = 1.
GTU	$C = 1 \ \& \ Z = 0$	Unsigned and greater.
EQ/Z	$Z = 1$	Equal/ Zero flag = 1.
N	$S = 1$	Negative.
LE	$(z = 1) \mid (S = 1 \ \& \ O = 0) \mid (S = 0 \ \& \ O = 1)$	Equal or signed and smaller.
O	$O = 1$	Overflow flag = 1.
GE	$(S = 1 \ \& \ O = 1) \mid (S = 0 \ \& \ O = 0)$	Equal or signed and greater.
LTU/NC	$C = 0$	Smaller/ Carry flag = 0.
LEU	$C = 0 \mid Z = 1$	Equal or smaller.
NE/NZ	$Z = 0$	Not equal/ Zero flag = 0.
PZ	$S = 0$	Positive or zero.
GT	$(S = 1 \ \& \ O = 1 \ \& \ Z = 0) \mid (S = 0 \ \& \ O = 0 \ \& \ Z = 0)$	Signed and greater.
NO	$O = 0$	Overflow flag = 0.
LT	$(S = 1 \ \& \ O = 0) \mid (S = 0 \ \& \ O = 1)$	Signed and smaller.

(If SB and FLG register status is as follows)

86

2.6.7 Sign-extension instruction

The sign-extension instruction extends bit length by substituting the sign bit for the bits to be extended. This section explains about the sign-extension instruction.

Sign-extension instruction

The sign-extension instruction comes in two types: a sign-extension instruction that extends bit length with the MSB (most significant bit), and a zero-extension instruction that extends bit length by forcibly filling the most significant bit with 0. When the size specifier ".B" is specified, the sign-extension instruction extends bit length to 16 bits; when the size specifier ".W" is specified, it extends bit length to 32 bits.

The zero-extension instruction extends bit length to 16 bits. An example of how the sign-extension instruction operates is shown in Figure 2.6.22.

Table 2.6.23 Sign-extension instruction

Mnemonic	Description Format	Explanation
EXTS	EXTS.B dest	Sign-extends dest to 16 bits.
	EXTS.W dest	Sign-extends dest to 32 bits.
	EXTS.B src,dest	Sign-extends src before transferring it to dest.
EXTZ	EXTZ src,dest	Zero-extends src to 16 bits before transferring it to dest.

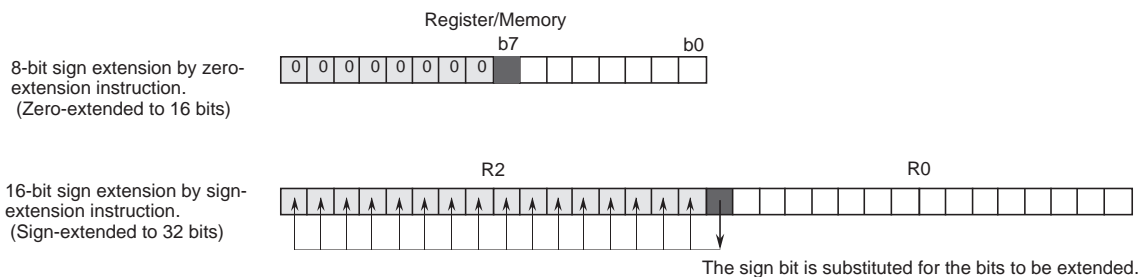


Figure 2.6.22 Typical operation of sign-extension instruction

2.6.8 Index instruction

The M16C/80 has index instructions to allow arrays to be referenced efficiently when programming in C language. Index instructions make it possible to specify array elements without address calculation. This section explains about index instructions.

Index instruction

The index instruction modifies addressing of the next instruction.

Index instructions classified by type are listed in Table 2.6.24. An example of how the index instruction operates is shown in Figure 2.6.23.

Table 2.6.24 Index instruction

Type	Function	
B BD BS	Modifies addressing of the next instruction in units of bytes.	
W WD WS	Modifies addressing of the next instruction in units of words.	
L LD LS	Modifies addressing of the next instruction in units of long words.	

Mnemonic	Description Format	Explanation
INDEXB	INDEXB.B src INDEXB.W src	The content of src of the INDEXB instruction is added to the address indicated by src, dest of the next instruction to be executed, with the sign not included, to find the effective address.
INDEXBD	INDEXBD.B src INDEXBD.W src	The content of src of the INDEXB instruction is added to the address indicated by dest (src in some instructions) of the next instruction to be executed, with the sign not included, to find the effective address.
INDEXBS	INDEXBS.B src INDEXBS.W src	The content of src of the INDEXB instruction is added to the address indicated by src of the next instruction to be executed, with the sign not included, to find the effective address.
INDEXW ^a	INDEXB.B src INDEXB.W src	Twice the content of src of the INDEXW instruction is added to the address indicated by src, dest of the next instruction to be executed, with the sign not included, to find the effective address.
INDEXWD ^a	INDEXWD.B src INDEXWD.W src	Twice the content of src of the INDEXB instruction is added to the address indicated by dest (src in some instructions) of the next instruction to be executed, with the sign not included, to find the effective address.
INDEXWS ^a	INDEXWS.B src INDEXWS.W src	Twice the content of src of the INDEXB instruction is added to the address indicated by src of the next instruction to be executed, with the sign not included, to find the effective address.
INDEXL ^b	INDEXL.B src INDEXL.W src	Four times the content of src of the INDEXW instruction is added to the address indicated by src, dest of the next instruction to be executed, with the sign not included, to find the effective address.
INDEXLD ^b	INDEXLD.B src INDEXLD.W src	Four times the content of src of the INDEXB instruction is added to the address indicated by dest (src in some instructions) of the next instruction to be executed, with the sign not included, to find the effective address.
INDEXLS ^b	INDEXLS.B src INDEXLS.W src	Four times the content of src of the INDEXB instruction is added to the address indicated by src of the next instruction to be executed, with the sign not included, to find the effective address.
BITINDEX	BITINDEX.B src BITINDEX.W src	The bit as many bits apart as indicated by src of the BITINDEX instruction from bit 0 at the address indicated by src or dest of the next instruction to be executed, is the target to be operated on.

^a: Corresponds to arrays arranged in units of words.

^b: Corresponds to arrays arranged in units of long words.

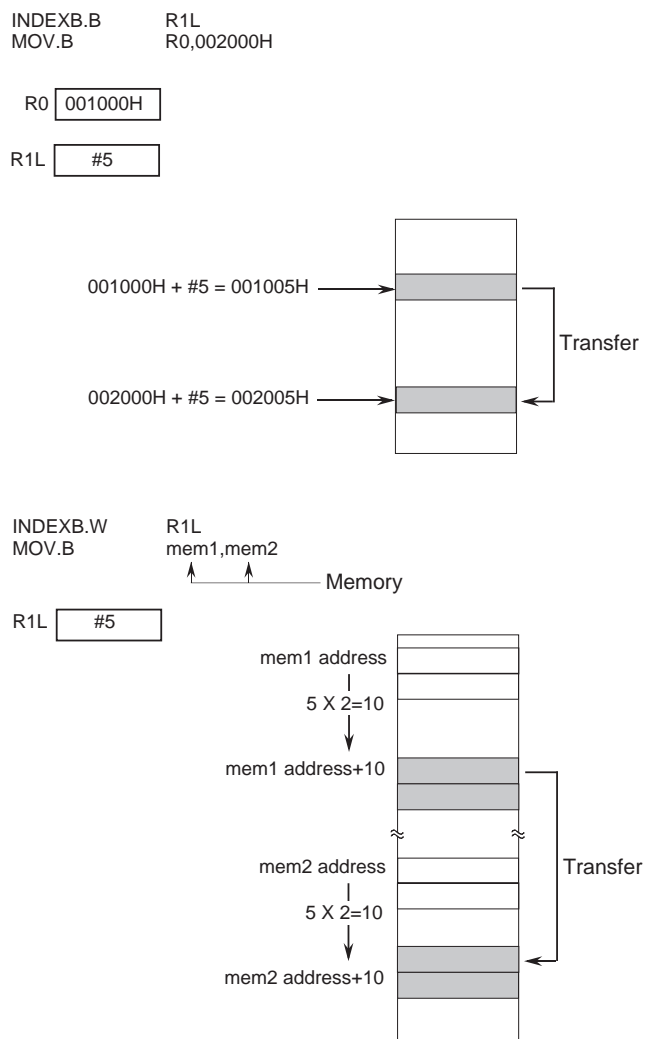


Figure 2.6.23 Typical operation of index instruction

2.6.9 High-level language and OS support instructions

The high-level language support instruction builds/frees a stack frame. The OS support instruction saves/restores task context. These instructions allow switching-over of complex processing or task context in high-level language to be executed by a single instruction.

Building Stack Frame

ENTER is an instruction to build a stack frame. Use #IMM to set bytes of the automatic variable area. Figure 2.6.24 shows an example of how this instruction works.

Table 2.6.25 Stack Frame Build Instruction

Mnemonic	Description Format	Explanation
ENTER	ENTER #IMM8	Builds stack frame.

ENTER #3

- 1) Saves FB register to stack area.
- 2) Transfers SP to FB.
- 3) Subtracts specified immediate from SP to modify SP (to allocate automatic variable area of called function).

[Before executing ENTER instruction]

[After executing ENTER instruction]

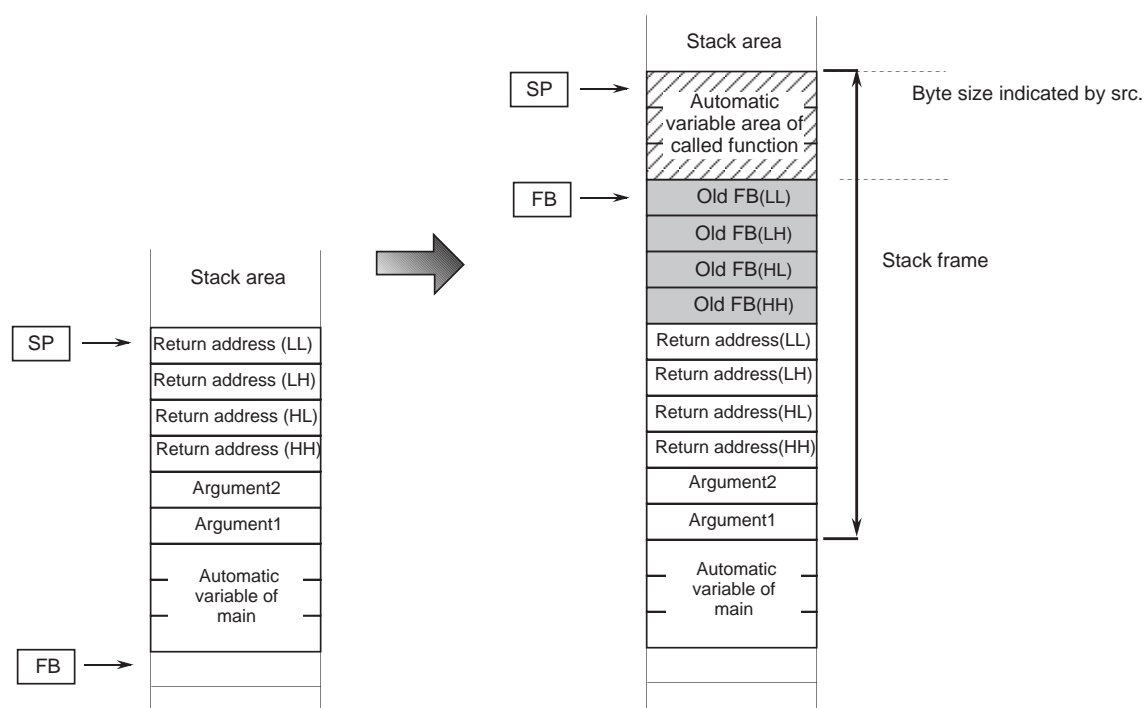


Figure 2.6.24 Typical operation of stack frame build instruction

Deallocate Stack Frame

The EXITD instruction deallocates the stack frame and returns control from the subroutine. It performs these operations simultaneously. Figure 2.6.25 shows an example of how the stack frame clean-up instruction works.

Table 2.6.26 Deallocate Stack Frame Instruction

Mnemonic	Description Format	Explanation
EXITD	EXITD	Deallocate stack frame.

EXITD

- 1) Transfers FB to SP.
- 2) Restores FB from stack area.
- 3) Returns from subroutine (function) (operates in the same way as RTS instruction).

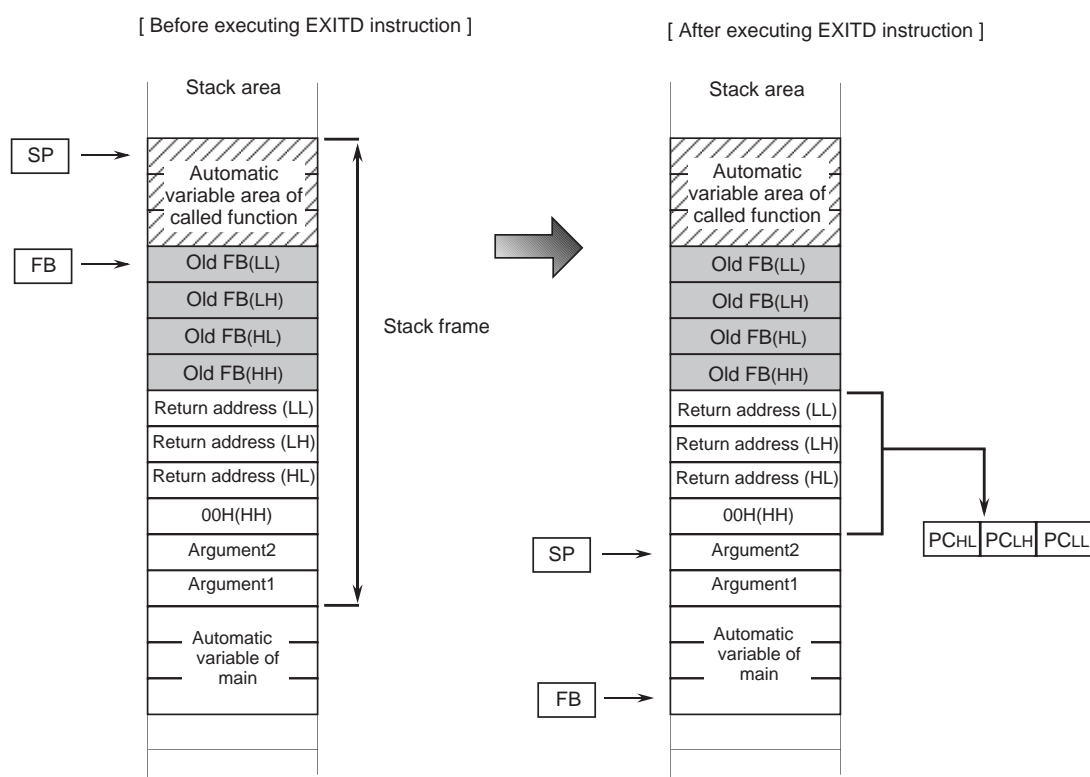


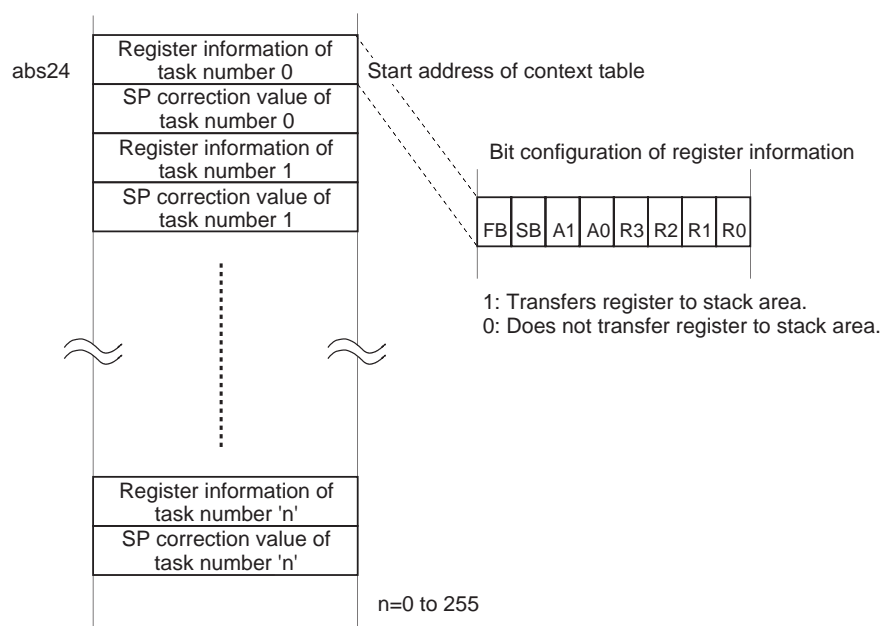
Figure 2.6.25 Typical operation of deallocate stack frame instruction

OS support instruction

The STCTX instruction saves task context. The LDCTX instruction restores task context. A table of task context is shown in Figure 2.6.26. The register information that is set in the context table shows the type of register to be saved as context in a stack area. The SP correction value shows the size in bytes of the register to be transferred. The OS support instruction uses these two pieces of information to save and restore task context to and from a stack area.

Table 2.6.27 OS Support Instructions

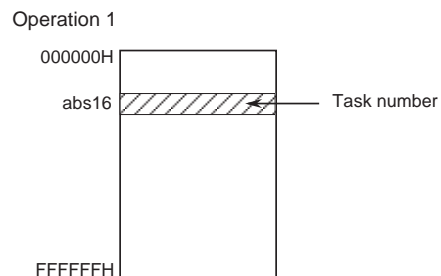
Mnemonic	Description Format	Explanation
STCTX	STCTX abs16,abs24 ^(Note)	Saves task context.
LDCTX	LDCTX abs16,abs24	Restores task context.

**Figure 2.6.26 Context table**

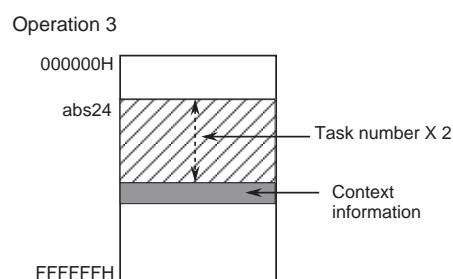
Note : abs16 is a memory address which stored task number (8 bits).
abs 24 is a top address of context table.

Operation for Saving Context (STCTX instruction)

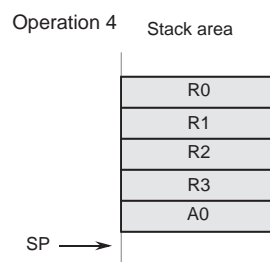
Operation 1
Reads memory content from the address indicated by abs16 as a task number (8-bit data).



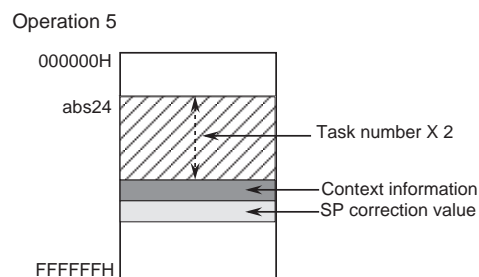
Operation 2
Doubles the value (task number) obtained in operation 1 and adds abs24 (start address of the context table).
 $(\text{task number}) \times 2 + \text{abs24}$



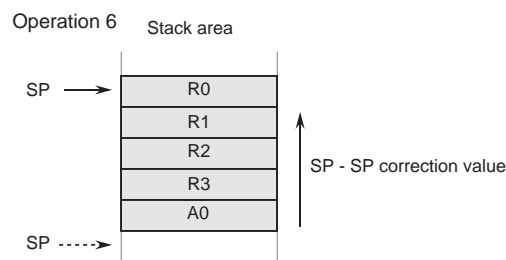
Operation 4
Saves the register specified by the context information to the stack area.



Operation 5
Reads memory content from the next address (incremented by 1) of the context information as an SP correction value (8-bit data).

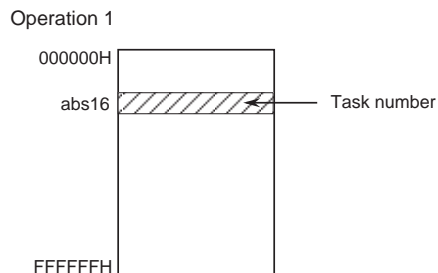


Operation 6
Subtracts the SP correction value from SP to correct the SP.

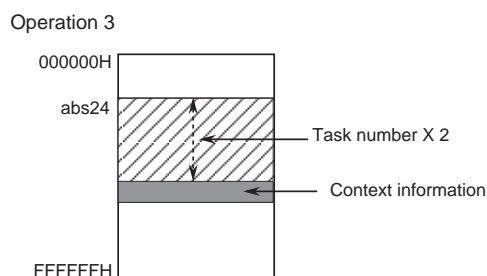


Operation for Restoring Context (LDCTX instruction)

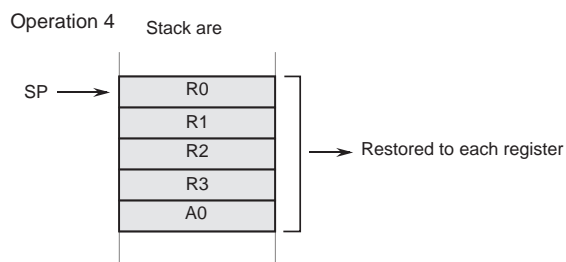
Operation 1
Reads memory content from the address indicated by abs16 as a task number (8-bit data).



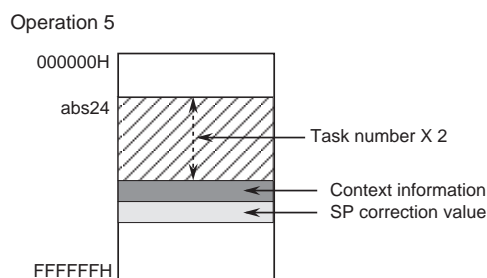
Operation 2
Doubles the value (task number) obtained in operation 1 and adds abs24 (start address of the context table).
(task number) x 2 + abs24



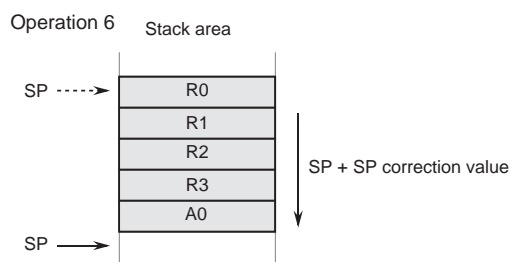
Operation 4
Restores the register specified by the context information from the stack area.
(At this time, the SP register value is not changed yet.)



Operation 5
Reads memory content from the next address (incremented by 1) of the context information as an SP correction value (8-bit data).



Operation 6
Subtracts the SP correction value from SP to correct the SP.



2.7 Outline of Interrupt

This section explains the types of interrupt sources available with the M16C/80 group and the internal processing (interrupt sequence) performed after an interrupt request is accepted until an interrupt routine is executed. For details on how to use each interrupt and how to set, refer to Chapter 4.

2.7.1 Interrupt Sources and Vector addresses

The following explains the interrupt sources available with the M16C/80 group.

Interrupt Sources in M16C/80 Group

Figure 2.7.1 shows the interrupt sources available with the M16C/80 group.

Hardware interrupts consist of six types of special interrupts such as reset and $\overline{\text{NMI}}$ and various peripheral I/O interrupts^(Note) that are dependent on built-in peripheral functions such as timers and external pins. Special interrupts are nonmaskable; peripheral I/O interrupts are maskable.

Maskable interrupts are enabled and disabled by an interrupt enable flag (I flag), an interrupt priority level select bit, and the processor interrupt priority level (IPL).

Software interrupts generate an interrupt request by executing a software interrupt instruction.

There are five types of software interrupts: INT instruction interrupt, BRK instruction interrupt, BRK2 instruction interrupt, overflow interrupt, and undefined instruction interrupt. Software interrupts are nonmaskable.

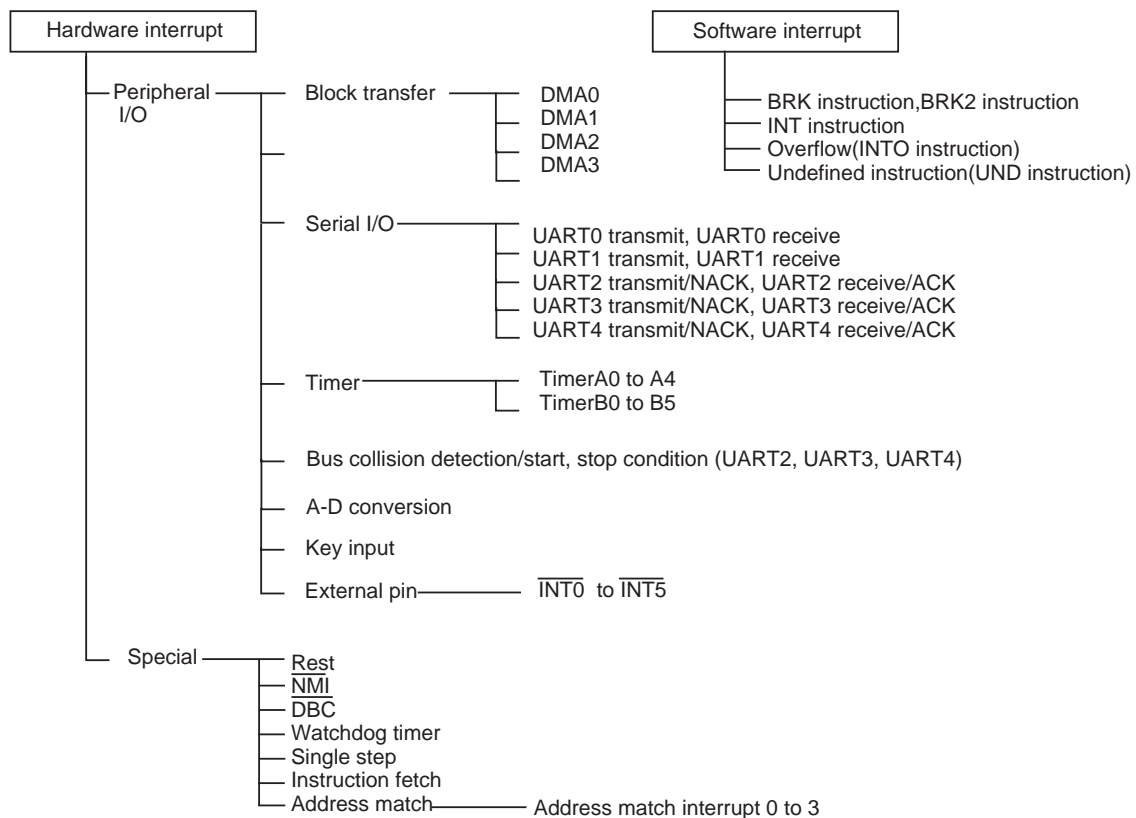


Figure 2.7.1 Interrupt sources in M16C/80 group

Note: Peripheral functions vary with each type of microcomputer used. For details about peripheral interrupts, refer to the data sheet and user's manual of your microcomputer.

Vector addresses

Figure 2.7.2 shows software interrupt and special interrupt vector addresses. Figure 2.7.3 shows hardware interrupt vector addresses. Before using these interrupts, set the start address of each relevant interrupt routine at these vector addresses.

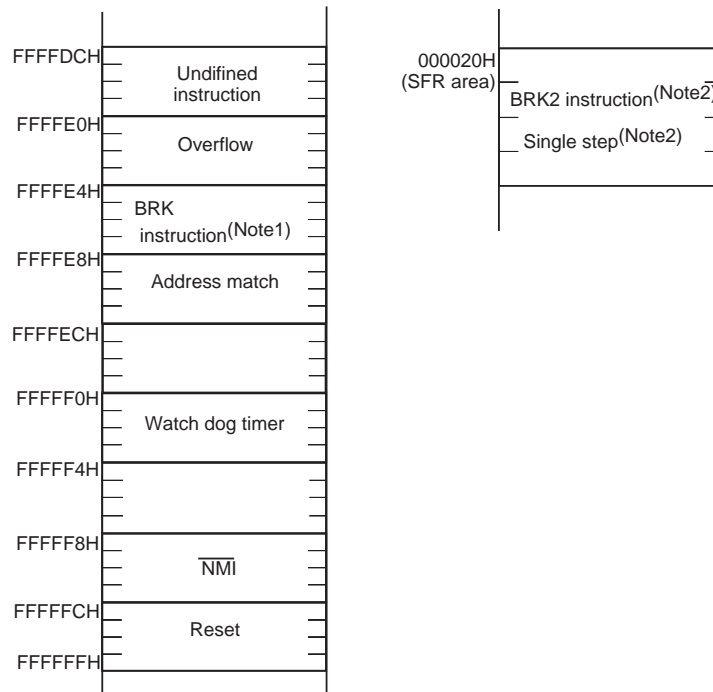


Figure 2.7.2 Software interrupt and special interrupt vector addresses

Note 1: If the vector contents all are "FFH," the program branches to the address indicated by the vector of software interrupt number 0 in the variable vector table.

Note 2: This area is inhibited against use by the user.

Software interrupt number	Vector table address Address (L) to address (H)	Interrupt source	Remarks
Software interrupt number 0	+0 to +3 (Note 1)	BRK instruction	Cannot be masked I flag
Software interrupt number 8	+32 to +35 (Note 1)	DMA0	
Software interrupt number 9	+36 to +39 (Note 1)	DMA1	
Software interrupt number 10	+40 to +43 (Note 1)	DMA2	
Software interrupt number 11	+44 to +47 (Note 1)	DMA3	
Software interrupt number 12	+48 to +51 (Note 1)	Timer A0	
Software interrupt number 13	+52 to +55 (Note 1)	Timer A1	
Software interrupt number 14	+56 to +59 (Note 1)	Timer A2	
Software interrupt number 15	+60 to +63 (Note 1)	Timer A3	
Software interrupt number 16	+64 to +67 (Note 1)	Timer A4	
Software interrupt number 17	+68 to +71 (Note 1)	UART0 transmit	
Software interrupt number 18	+72 to +75 (Note 1)	UART0 receive	
Software interrupt number 19	+76 to +79 (Note 1)	UART1 transmit	
Software interrupt number 20	+80 to +83 (Note 1)	UART1 receive	
Software interrupt number 21	+84 to +87 (Note 1)	Timer B0	
Software interrupt number 22	+88 to +91 (Note 1)	Timer B1	
Software interrupt number 23	+92 to +95 (Note 1)	Timer B2	
Software interrupt number 24	+96 to +99 (Note 1)	Timer B3	
Software interrupt number 25	+100 to +103 (Note 1)	Timer B4	
Software interrupt number 26	+104 to +107 (Note 1)	$\overline{\text{INT5}}$	
Software interrupt number 27	+108 to +111 (Note 1)	$\overline{\text{INT4}}$	
Software interrupt number 28	+112 to +115 (Note 1)	$\overline{\text{INT3}}$	
Software interrupt number 29	+116 to +119 (Note 1)	$\overline{\text{INT2}}$	
Software interrupt number 30	+120 to +123 (Note 1)	$\overline{\text{INT1}}$	
Software interrupt number 31	+124 to +127 (Note 1)	$\overline{\text{INT0}}$	
Software interrupt number 32	+128 to +131 (Note 1)	Timer B5	
Software interrupt number 33	+132 to +135 (Note 1)	UART2 transmit/NACK (Note 2)	
Software interrupt number 34	+136 to +139 (Note 1)	UART2 receive/ACK (Note 2)	
Software interrupt number 35	+140 to +143 (Note 1)	UART3 transmit/NACK (Note 2)	
Software interrupt number 36	+144 to +147 (Note 1)	UART3 receive/ACK (Note 2)	
Software interrupt number 37	+148 to +151 (Note 1)	UART4 transmit/NACK (Note 2)	
Software interrupt number 38	+152 to +155 (Note 1)	UART4 receive/ACK (Note 2)	
Software interrupt number 39	+156 to +159 (Note 1)	Bus collision detection, start/stop condition detection (UART2) (Note 2)	
Software interrupt number 40	+160 to +163 (Note 1)	Bus collision detection, start/stop condition detection (UART3) (Note 2)	
Software interrupt number 41	+164 to +167 (Note 1)	Bus collision detection, start/stop condition detection (UART4) (Note 2)	
Software interrupt number 42	+168 to +171 (Note 1)	A-D	
Software interrupt number 43	+172 to +175 (Note 1)	Key input interrupt	
Software interrupt number 44 to Software interrupt number 63	+176 to +179 (Note 1) to +252 to +255 (Note 1)	Software interrupt	Cannot be masked I flag

Note 1: Address relative to address in interrupt table register (INTB).

Note 2: When I²C mode is selected, NACK/ACK, start/stop condition detection interrupts are selected. The fault error interrupt is selected when SS pin is selected.

Figure 2.7.3 Hardware interrupt vector addresses

2.7.2 Variable vector table

The variable vector table is a 256-byte vector table that starts from the address indicated by the interrupt table register (INTB). (See Figure 2.7.3.) The vector table can be located in any area except the SFR area and the extended reserved area.

Variable vector table

One vector consists of 4 bytes, with each vector assigned software interrupt numbers 0 to 63. Using the INT instruction and a software interrupt number, it is possible to execute a peripheral I/O interrupt routine in a simulated manner. Figure 2.7.4 shows how the variable vector table is located in memory.

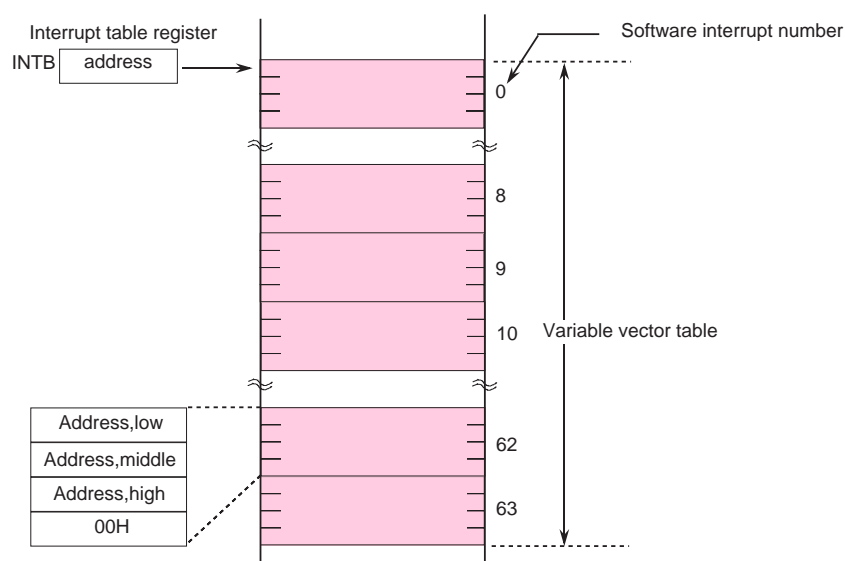


Figure 2.7.4 Example of how the variable vector table is located

2.7.3 Interrupt generation conditions and interrupt control register bit configuration

This section describes conditions under which an interrupt is accepted and the bit configuration of the interrupt control register.

Interrupt generation conditions

When an interrupt is requested, it is accepted when all of the following three conditions are met:

- (1) Interrupt enable flag (I flag) = 1 (interrupt enabled)
- (2) Processor interrupt level (IPL) < Interrupt priority level of the requested interrupt
- (3) Interrupt request bit (interrupt control register bit 3) = 1

Bit configuration of the interrupt control register

Figure 2.7.5 shows the interrupt control register provided for each interrupt.

Interrupt control register

	Symbol	Address	When reset
	ADIC	0073 ₁₆	XXXXX0002
	BCNiIC(i=2 to 4)	008F ₁₆ , 0071 ₁₆ , 0091 ₁₆	XXXXX0002
	DMiIC(i=0 to 3)	0068 ₁₆ , 0088 ₁₆ , 006A ₁₆ , 008A ₁₆	XXXXX0002
	KUPIC	0093 ₁₆	XXXXX0002
	TAiIC(i=0 to 4)	006C ₁₆ , 008C ₁₆ , 006E ₁₆ , 008E ₁₆ , 0070 ₁₆	XXXXX0002
	TBiIC(i=0 to 5)	0094 ₁₆ , 0076 ₁₆ , 0096 ₁₆ , 0078 ₁₆ , 0098 ₁₆ , 0069 ₁₆	XXXXX0002
	SiTiC(i=0 to 4)	0090 ₁₆ , 0092 ₁₆ , 0089 ₁₆ , 008B ₁₆ , 008D ₁₆	XXXXX0002
	SiRIC(i=0 to 4)	0072 ₁₆ , 0074 ₁₆ , 006B ₁₆ , 006D ₁₆ , 006F ₁₆	XXXXX0002

b7b6b5b4b3b2b1b0

Symbol	Address	When reset
INTiIC(i=0 to 5)	009E ₁₆ , 007E ₁₆ , 009C ₁₆ , 007C ₁₆ , 009A ₁₆ , 007A ₁₆	XX00X0002

Bit symbol	Bit name	Function	R	W
ILVL0	Interrupt priority level select bit	b2 b1 b0 0 0 0 : Level 0 (interrupt disabled) 0 0 1 : Level 1 0 1 0 : Level 2 0 1 1 : Level 3 1 0 0 : Level 4 1 0 1 : Level 5 1 1 0 : Level 6 1 1 1 : Level 7	○	○
ILVL1			○	○
ILVL2			○	○
IR			○	○ (Note 1)
POL	Polarity select bit	0 : Selects falling edge or L level 1 : Selects rising edge or H level	○	○
LVS	Level sense/edge sense select bit	0 : Edge sense 1 : Level sense (Note 3)	○	○
Nothing is assigned. When write, set "0". When read, their contents are indeterminate.			—	—

Note 1: This bit can only be accessed for reset (= 0), but cannot be accessed for set (= 1).

Note 2: When INT3 to INT5 are used for data bus in microprocessor mode or memory expansion mode, set the interrupt disabled to INT3IC, INT4IC and INT5IC.

Note 3: When level sense is selected, set related bit of interrupt cause select register (address 031F₁₆) to one edge.

Figure 2.7.5 Bit configuration of the interrupt control register

Note 3: The symbols shown here are for the M16C/80 group. They vary with each microcomputer type used.

2.7.4 Interrupt acceptance timing and sequence

This section describes the interrupt acceptance timing and interrupt sequence.

Interrupt acceptance timing

When an interrupt request occurs while executing an instruction, the priority of the requested interrupt is resolved after the instruction being executed finishes, and an interrupt sequence begins in the next cycle. The interrupt acceptance timing in this case is shown in Figure 2.7.6. However, if an interrupt request occurs when executing a string instruction (SCMPU, SIN, SMOVB, SMOVF, SMOVU, SSTR, or SOUT) or multiply/accumulate instruction (RMPA), the instruction being executed is suspended and an interrupt sequence is entered. The interrupt acceptance timing in this case is shown in Figure 2.7.7.

1. Interrupt under normal condition

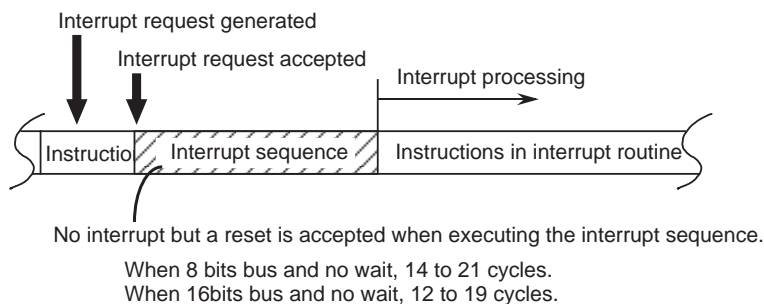


Figure 2.7.6 Interrupt acceptance timing 1

2. Interrupt under exceptional condition

If an interrupt request is generated when executing one of the following instructions, the interrupt sequence occurs in the middle of that instruction execution.

- (1) String transfer instruction (SCMPU, SIN, SMOVB, SMOVF, SMOVU, SSTR, SOUT)
- (2) Sum-of-product calculating instruction (RMPA)

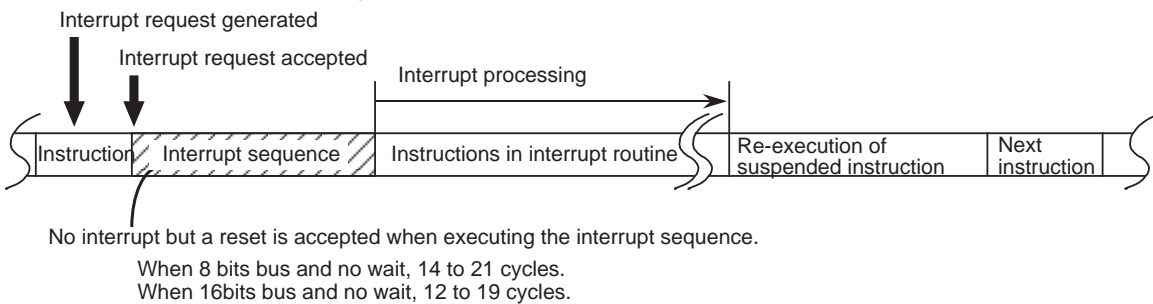


Figure 2.7.7 Interrupt acceptance timing 2

Interrupt sequence

The following explains an interrupt sequence from when an interrupt request is accepted to when an interrupt routine is executed.

- (1) The CPU reads address 000000H (or 000002H for fast interrupts) to get interrupt information (interrupt number, interrupt request level). The relevant interrupt request bit is then reset to 0.
- (2) The content of the flag register (FLG) immediately before the interrupt sequence begins is saved to an internal temporary register^(Note) of the CPU.
- (3) The interrupt enable flag (I flag), debug flag (D flag), and stack pointer specification flag (U flag) are reset to 0. (However, the U flag does not change if an INT instruction of software interrupt numbers 32 to 63 was being executed when the interrupt occurred.) Thus, by the above operations...
 - (a) The stack pointer is forcibly made the interrupt stack pointer (ISP). (However, if an INT instruction of software interrupt numbers 32 to 63 was being executed when the interrupt occurred, the stack pointer (ISP or USP) that was active when the interrupt occurred is used.)
 - (b) Multiple interrupts are disabled.
 - (c) Single-step interrupt is disabled.
- (4) The content of the CPU's internal temporary register^(Note) and that of the program counter (PC) are saved to the stack area. For fast interrupts, they are saved to the save flag register (SVF) and save PC register (SVP).
- (5) The interrupt priority level of the accepted interrupt is set in the processor interrupt priority level (IPL).

When the interrupt sequence is completed, instructions are executed beginning with the start address of the interrupt routine.

Note: This register cannot be used by the user.

2.7.5 Interrupt priority

This section explains about interrupt priority.

Interrupt priority

If two or more interrupt requests simultaneously are sampled active (= asserted), the interrupt with the highest priority among those interrupts is accepted. Maskable interrupts (peripheral I/O interrupts) can be assigned any desired priority by using the interrupt priority level select bits. However, when requested interrupts have the same priority level, the first interrupt requested is accepted, and the remaining other interrupts are accepted according to the priority^(Note) that is set in hardware.

Nonmaskable interrupts such as a reset (handled as an interrupt of the highest priority) and a watchdog timer interrupt have their priorities set in hardware. The interrupt priorities set in hardware are shown in Figure 2.7.8.

Software interrupts are unaffected by interrupt priority. When an instruction is executed, the program always branches to the relevant interrupt routine.

Note: This priority varies with each type of microcomputer. Be sure to consult data sheets and user's manual.

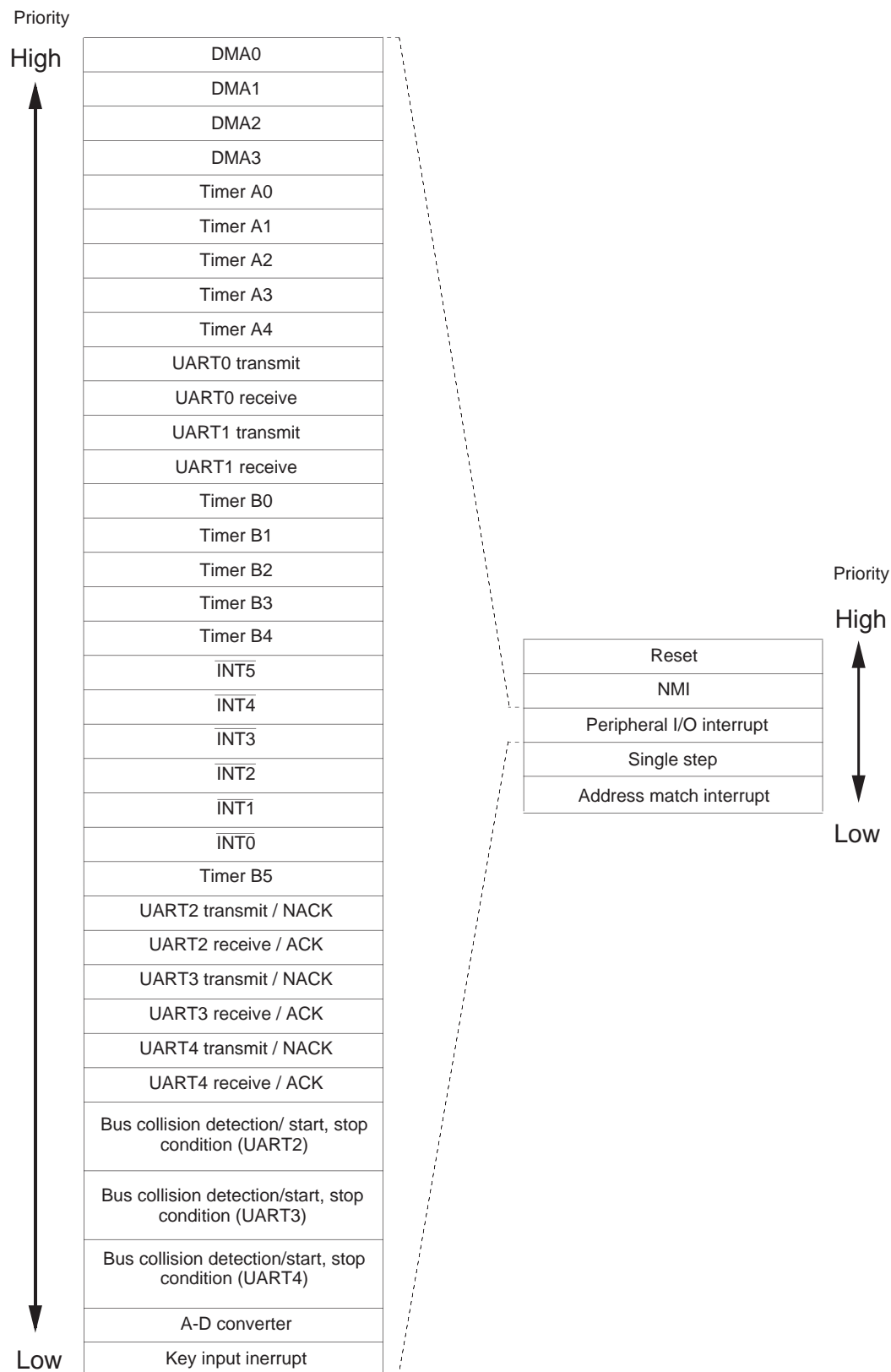


Figure 2.7.8 Interrupt priority by determined by hardware

Chapter 3

Functions of Assembler

- 3.1 Outline of AS308 System
- 3.2 Method for Writing Source Program

3.1 Outline of AS308 System

The AS308 system is a software system that supports development of programs for controlling the M16C/80 series single-chip microcomputers at the assembly language level. In addition to the assembler, the AS308 system comes with a linkage editor and a load module converter.

This section explains the outline of AS308.

Functions

- Relocatable assemble function
- Optimized code generating function
- Macro function
- High-level language source level debug function
- Various file generating function
- IEEE-695 format^(Note 1) file generating function

Configuration

The AS308 system consists of the following programs:

- **Assembler driver (as308)**

This is an execution file to start up the macroprocessor and assembler processor. This assembler driver can process multiple assembly source files.

- **Macroprocessor (mac308)**

This program processes macro directive commands in the assembly source file and performs preprocessing for the assembler processor, thereby generating an intermediate file. This intermediate file is erased after processing by the assembler processor is completed.

- **Assembler processor (asp308)**

This program converts the intermediate file generated by the macroprocessor into a relocatable module file.

- **Linkage editor (ln308)**

This program links the relocatable module files generated by the assembler processor to generate an absolute module file.

- **Load module converter (lmc308)**^(Note 2)

This program converts the absolute module file generated by the linkage editor into a machine language file that can be programmed into ROM.

- **Librarian (lb308)**

By reading in the relocatable module files, this program generates and manages a library file.

- **Cross referencer (xrf308)**

This program generates a cross reference file that contains definition of various symbols and labels used in the assembly source file created by the user.

- **Absolute lister (abs308)**

Based on the address information in the absolute module file, this program generates an absolute list file that can be output to a printer.

Note 1: IEEE stands for the Institute of Electrical and Electronics Engineers.

Note 2: The load module converter is a program to convert files into the format in which they can be programmed into M16C/80 series ROMs.

Figure 3.1.1 schematically shows the assemble processing performed by the AS308 system.



Input/output Files Handled by AS308

The table below separately lists the input files and the output files handled by the AS308 system. Any desired file names can be assigned. However, if the extension of a file name is omitted, the AS308 system automatically adds a default file extension. These default extensions are shown in parenthesis in the table below.

Table 3.1.1 List of Input/output Files

Program Name	Input File Name (Extension)	Output File Name (Extension)
Assembler as308	Source file(.as30) Include file(.inc)	Relocatable module file(.r30) Assembler list file(.lst) Assembler error tag file(.atg)
Linkage editor ln308	Relocatable module file(.r30) Library file(.lib)	Absolute module file(.x30) Map file(.map) Link error tag file(.ltg)
Load module converter lmc308	Absolute module file(.x30)	Motorola S format file(.mot) Extended Intel HEX format file(.hex)
Librarian lb308	Relocatable module file(.r30) Library file(.lib)	Library file(.lib) Relocatable module file(.r30) Library list file(.lls)
Cross referencer xrf308	Assemble source file(.a30) Assembler list file(.lst)	Cross reference file(.xrf)
Absolute lister abs308	Absolute module file(.x30) Assembler list file(.lst)	Absolute list file(.als)

3.2 Method for Writing Source Program

This section explains the basic rules, address control, and directive commands that need to be understood before writing the source programs that can be processed by the AS308 system. For details about the AS308 system itself, refer to AS308 User's Manuals, "Operation Part" and "Programming Part".

3.2.1 Basic Rules

The following explains the basic rules for writing the source programs to be processed by the AS308 system.

Precautions on Writing Programs

Pay attention to the following precautions when writing the source programs to be processed by the AS308 system:

- Do not use the AS308 system reserved words for names in the source program.
- Do not use a character string consisting of one of the AS308 system directive commands with the period removed, because such a character string could affect processing by AS308. They can be used in names without causing an error.
- Do not use system labels (the character strings that begin with ..) because they may be used for future extension of the AS308 system. When they are used in the source program created by the user, the assembler does not output an error.

Character Set

The characters listed below can be used to write the assembly program to be processed by the AS308 system.

Uppercase English alphabets

A B C D E F G H I J K L M N O P Q R
S T U V W X Y Z

Lowercase English alphabets

a b c d e f g h i j k l m n o p q r s t u
v w x y z

Numerals

0 1 2 3 4 5 6 7 8 9

Special characters

" # % & ' () * + , - . / : ; [\] ^ _ | ~

Blank characters

(space) (tab)

New line characters

(return) (line feed)

Reserved Words

The following lists the reserved words of the AS308 system. The reserved words are not discriminated between uppercase and lowercase. Therefore, "abs", "ABS", "Abs", "ABs", "AbS", "abS", "aBs", "aBS" — all are the same as the reserved word "ABS".

Mnemonic

ABS	ADC	ADCF	ADD	ADDX	ADJNZ	AND
BAND	BCLR	BITINDEX	BMC	BMEQ	BMGE	BMGEU
BMGT	BMGTU	BMLE	BMLEU	BMLT	BMLTU	BMN
BMNC	BMNE	BMNO	BMNZ	BMO	BMPZ	BMZ
BNAND	BNOR	BNOT	BNTST	BNXOR	BOR	BRK
BRK2	BSET	BTST	BTSTC	BTSTS	BXOR	CLIP
CMP	CMPX	DADC	DADD	DEC	DIV	DIVU
DIVX	DSBB	DSUB	ENTER	EXITD	EXTS	EXTZ
FCLR	FREIT	FSET	INC	INDEXB	INDEXBD	INDEXBS
INDEXW	INDEXWD	INDEXWS	INDEXL	INDEXLD	INDEXLS	INT
INTO JC	JEQ	JGE	JGEU	JGT	JGTU	
JLE	JLEU	JLT	JLTU	JMP	JMPI	JMPS
JN	JNC	JNE	JNO	JNZ	JO	JPZ
JSR	JSRI	JSRS	JZ	LDC	LDCTX	LDIPL
MAX	MIN	MOV	MOVA	MOVHH	MOVHL	MOVLH
MOVLL	MUL	MULEX	MULU	NEG	NOP	NOT
OR	POP	POPC	POPM	PUSH	PUSHA	PUSHC
PUSHM	REIT	RMPA	ROLC	RORC	ROT	RTS
SBB	SBJNZ	SCC	SCEQ	SCGE	SCGEU	SCGT
SCGTU	SCLE	SCLEU	SCLT	SCLTU	SCMPU	SCN
SCNC	SCNE	SCNO	SCNZ	SCO	SCPZ	SCZ
SHA	SHL	SIN	SMOVB	SMOVF	SMOVU	SOUT
SSTR	STC	STCTX	STNZ	STZ	STZX	SUB
SUBX	TST	UND	WAIT	XCHG	XOR	

Register/flag

A0	A1	B	C	D	DCT0	DCT1
DMA0	DMA1	DMD0	DMD1	DRA0	DRA1	DRC0
DRC1	DRA0	DSA1	FB	FLG	I	INTB
IPL	ISP	O	PC	R0	R0H	R0L
R1	R1H	R1L	R2	R2R0	R3	R3R1
S	SB	SP	SVF	SVP	U	USP
VCT	Z					

Operators

SIZEOF TOPOF

System Label(all names that begin with Two Periods "..")

Description of Names

Any desired names can be used in the source program as defined.
Names can be divided into the following five types. Note that the AS308 system reserved words cannot be used in names.^(Note)

- (1) Label
- (2) Symbol
- (3) Bit symbol
- (4) Location symbol
- (5) Macro name

Rules for writing names

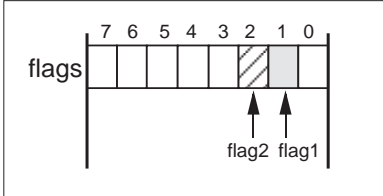
- (1) Names can be written using alphanumeric characters and "_" (underscore). Each name must be within 255 characters in length.
- (2) Names are case-sensitive, so they are discriminated between uppercase and lowercase.
- (3) Numerals cannot be used at the beginning of a name.

Note: Program operation cannot be guaranteed if any reserved word is used.

Types of Names

Table 3.2.1 shows the method for defining names.

Table 3.2.1 Types of Names Defined by User

Label	Symbol
<p>Function Indicates a specific memory address.</p> <p>Definition method Always add ":" (colon) at the end of each name. There are two methods of definition.</p> <ol style="list-style-type: none"> 1. Allocate an area with a directive command. Example: <pre>flag: .BLKB 1 work: .BLKB 1</pre> 2. Write a name at the beginning of a source line. Example: <pre>name1: _name: sum_name:</pre> <p>Reference method Write the name in the operand of an instruction. Example: <pre>MP sym_name</pre> </p>	<p>Function Indicates a constant value.</p> <p>Definition method Use a directive command that defines a numeral. Example: <pre>value1 .EQU 1 value2 .EQU 2</pre> </p> <p>Reference method Write a symbol in the operand of an instruction. Example: <pre>MOV.W R0,value2+1 value3 .EQU value2+1</pre> </p>
Bit symbol	Location symbol
<p>Function Indicates a specific bit position in specific memory.</p> <p>Definition method Use a directive command that defines a bit symbol. Example: <pre>flag1 .BTEQU 1,flags flag2 .BTEQU 2,flags flag3 .BTEQU 20,flags</pre> </p>  <p>Reference method The bit symbol can be written in the operand of a single-bit manipulating instruction. Example: <pre>BCLR flag1 BCLR flag2 BCLR flag3</pre> </p>	<p>Function Indicates the current line of the source program.</p> <p>Definition method Unnecessary.</p> <p>Reference method Simply write a dollar mark (\$) in the operand to indicate the address of the line where it is written. Example: <pre>JMP \$+5</pre> </p>

Description of Operands

For mnemonics and directive commands, write an operand to indicate the subject to be operated on by that instruction. Operands are classified into five types by the method of description. Some instructions do not have an operand. For details about use of operands in instructions and types of operands, refer to explanation of the method for writing each instruction.

- **Numeric value**

Numeric values can be written in decimal, hexadecimal, binary, and octal. Table 3.2.2 shows types of operands, description examples, and how to write the operand.

Table 3.2.2 Description of Operands

Type	Description Example	Method of Description
Binary	10010001B 10010001b	Write 'B' or 'b' at the end of the operand.
Octal	60702o 60702O	Write 'O' or 'o' at the end of the operand.
Decimal	9423	Do not write anything at the end of the operand.
Hexadecimal	0A5FH 5FH 0a5fh 5fh	Use numerals 0 to 9 and alphabets 'a' to 'f' or 'A' to 'F' to write the operand and add 'H' or 'h' at the end. However, if the operand value begins with an alphabet, add '0' at the beginning.
Floating-point number	3.4E35 3.4E-35 -.5e20 5e20	Write an exponent including the sign after 'E' or 'e' in the exponent part. For 3.4×10^{35} , write 3.4E35.
Name	loop	Write a label or symbol name directly as it is.
Expression	256/2 label/3	Use a numeric value, name, and operator in combination to write an expression.
Character string	"string" 'string'	Enclose a character string with single or double quotations when writing it.

- Floating-point number

Numeric values within the range shown below that are represented by floating-point numbers can be written in the operand of an instruction. The method for writing floating-point numbers and description examples are shown in Table 3.2.2 in the preceding page. Floating-point numbers can only be used in the operands of the directive commands ".DOUBLE" and ".FLOAT". Table 3.2.3 lists the range of values that can be written in each of these directive commands.

Table 3.2.3 Description Range of Floating-point Numbers

Directive Command	Description Range
FLOAT (32 bits long)	$1.17549435 \times 10^{-38}$ to $3.40282347 \times 10^{38}$
DOUBLE (64 bits long)	$2.2250738585072014 \times 10^{-308}$ to $1.7976931348623157 \times 10^{308}$

- Name

Label and symbol names can be written in the operand of an instruction. The method for writing names and a description example are shown in Table 3.2.2 in the preceding page.

- Expression

An expression consisting of a combination of a numeric value, name, and operator can be written in the operand of an instruction. A combination of multiple operators can be used in an expression. When writing an expression as a symbol value, make sure that the value of the expression will be fixed when the program is assembled. The value that derives from calculation of an expression is within the range of -2,147,483,648 to 2,147,483,648. Floating-point numbers can be used in an expression. The method for writing expressions and description examples are shown in Table 3.2.2 in the preceding page.

- Character string

A character string can be written in the operand of some directive commands. Use 7-bit ASCII code to write a character string. Enclose a character string with single or double quotations when writing it. The method for writing character strings and description examples are shown in Table 3.2.2 in the preceding page.

Operator

Table 3.2.4 lists the operators that can be written in the source programs for AS308.

Table 3.2.4 List of Operators

Monadic operators		Conditional operators	
+	Positive value	>	Left-side value is greater than right-side value
-	Negative value	<	Right-side value is greater than left-side value
~	NOT	>=	Left-side value is equal to or greater than right-side value
SIZEOF	Section size (in bytes)	<=	Right-side value is equal to or greater than left-side value
TOPOF	Start address of section	==	Left-side value and right-side value are equal
Dyadic operators		!=	Left-side value and right-side value are not equal
		Calculation priority modifying operator	
+	Add	()	A term enclosed with () is calculated before any other term. If multiple terms in an expression are enclosed with (), the leftmost term has priority. Parentheses () can be nested.
-	Subtract		
*	Multiply		
/	Divide		
%	Remainder		
>>	Shift bits right		
<<	Shift bits left		
&	AND		
	OR		
^	Exclusive OR		

Note 1: For operators "SIZEOF" and "TOPOF," be sure to insert a space or tag between the operator and operand.
Note 2: Conditional operators can only be written in the operands of directive commands ".IF" and ".ELIF".

Calculation Priority

Calculation is performed in order of priorities of operators beginning with the highest priority operator. Table 3.2.5 lists the priorities of operators. If operators in an expression have the same priority, calculation is performed in order of positions from left to right. The priority of calculation can be changed by enclosing the desired term in an expression with ().

Table 3.2.5 Calculation Priority

Priority Level	Type of Operator	Content
High ↑	1	Calculation priority modifying operator (,)
	2	Monadic operator 1 + , -, ~ , SIZEOF , TOPOF
	3	Dyadic operator 1 * , / , %
	4	Dyadic operator 2 + , -
↓ Low	5	Dyadic operator 3 >> , <<
	6	Dyadic operator 4 &
	7	Dyadic operator 5 , ^
	8	Conditional operator > , < , >= , <= , == , !=

Description of Lines

AS308 processes the source program one line at a time. Lines are separated by the new line character. A section from a character immediately after the new line character to the next new line character is assumed to be one line. The maximum number of characters that can be written in one line is 255. Lines are classified into five types by the content written in the line. Table 3.2.6 shows the method for writing each type of line.

- Directive command line
- Assembly source line
- Label definition line
- Comment line
- Blank line

Table 3.2.6 Types of Lines

Directive Command Line	Assembly Source Line
<p>Function This is the line in which as30 directive command is written.</p> <p>Description method Only one directive command can be written in one line. A comment can be written in the directive command line.</p> <p>Precautions No directive command can be written along with a mnemonic in the same line.</p> <p>Example:</p> <pre> sym .SECTION program,DATA work: .ORG 00H .EQU 0 .BLKB 1 .ALIGN .PAGE "newpage" .ALIGN </pre>	<p>Function This is the line in which a mnemonic is written.</p> <p>Description method A label name (at beginning) and a comment can be written in the assembly source line.</p> <p>Precautions Only one mnemonic can be written in one line. No mnemonic can be written along with a directive command in the same line.</p> <p>Example:</p> <pre> MOV.W #0,R0 main: RTS MOV.W #0,A0 RTS </pre>
Label Definition Line	Comment Line
<p>Function This is the line in which only a label name is written.</p> <p>Description method Always be sure to write a colon (:) immediately following the label name.</p> <p>Example:</p> <pre> start: label: .BLKB 1 main: nop loop: </pre>	<p>Function This is the line in which only a comment is written.</p> <p>Description method Always be sure to write a semicolon (;) before the comment.</p> <p>Example:</p> <pre> ; Comment line MOV.W #0,A0 </pre>
	Blank Line
	<p>Function This is the line in which no meaningful character is written.</p> <p>Description method Write only a space, tab, or new line code in this line.</p>

3.2.2 Address Control

The following explains the AS308 system address control method.

The AS308 system does not take the RAM and ROM sizes into account as it controls memory addresses. Therefore, consider the actual address range in your application when writing the source programs and linking them.

Method of Address Control

The AS308 system manages memory addresses in units of sections. The division of each section is defined as follows. Sections cannot be nested as they are defined.

Division of section

- (1) An interval from the line in which directive command ".SECTION" is written to the line in which the next directive command ".SECTION" is written
- (2) An interval from the line in which directive command ".SECTION" is written to the line in which directive command ".END" is written

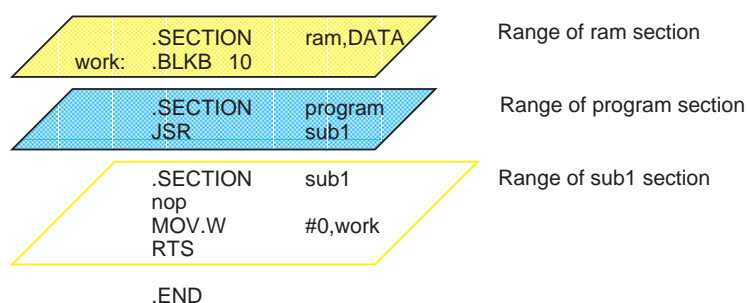


Figure 3.2.1 Range of sections in AS308 system

Types of Sections

A type can be set for sections in which units memory addresses are managed. The instructions that can be written in a section vary with each type of section.

Table 3.2.7 Types of Sections

Type	Content and Description Example
CODE (program area)	<ul style="list-style-type: none">• This is an area where the program is written.• All instructions except some directive commands that allocate memory can be written in this area.• CODE-type sections must be specified in the absolute module that they be located in the ROM area. Example: .SECTION program,CODE
DATA (data area)	<ul style="list-style-type: none">• This is an area where memory whose contents can be changed is located.• Directive commands that allocate memory can be written in this area.• DATA-type sections must be specified in the absolute module that they be located in the RAM area. Example: .SECTION mem,DATA
ROMDATA (fixed data area)	<ul style="list-style-type: none">• This is an area where fixed data other than the program is written.• ROMDATA-type sections must be specified in the absolute module that they be located in the ROM area. Example: .SECTION const,ROMDATA

Section Attribute

A section in which units memory addresses are controlled is assigned its attribute when assembling the program.

Table 3.2.8 Section Attributes

Attribute	Content and Description Example
Relative	<ul style="list-style-type: none">• Addresses in the section become relocatable values when the program is assembled.• The values of labels defined in the relative attribute section are relocatable.
Absolute	<ul style="list-style-type: none">• Addresses in the section become absolute values when the program is assembled.• The values of labels defined in the absolute attribute section are absolute.• To make a section assume the absolute attribute, specify the address with directive command ".ORG" in the line next to one where directive command ".SECTION" is written. <p>Example: .SECTION program,DATA .ORG 1000H</p>

Specifying an even address for the start address

For relative attribute sections, the start address of the section that is determined when linking can be set to be always located at an even address. If this adjustment is desired, specify "ALIGN" for the operand of the directive command ".SECTION."

Example:

```
.section    program,CODE,ALIGN
```

Address Control by AS308 System

The following shows how an assembly source program written in multiple files is converted into a single execution format file.

Address control by as308

- (1) For sections that will be assigned the absolute attribute, the assembler determines absolute addresses sequentially beginning with a specified address.
- (2) For sections that will be assigned the relative attribute, the assembler determines addresses sequentially for each section beginning with 0. The start address of all relative attribute sections are 0.

Address control by ln308

- (1) Sections of the same name in all files are arranged in order of specified files.
- (2) The start address of sections thus rearranged is determined as specified by the command option (-order) of ln308.
- (3) The start address of the first section is determined sequentially beginning with 0 unless otherwise specified.
- (4) Sections with different names are located at contiguous addresses in the order they are loaded into ln308 unless otherwise specified.
- (5) When an attempt is made to locate an absolute attribute section after a relative attribute section of the same name, an error results.

Address values determined by as308

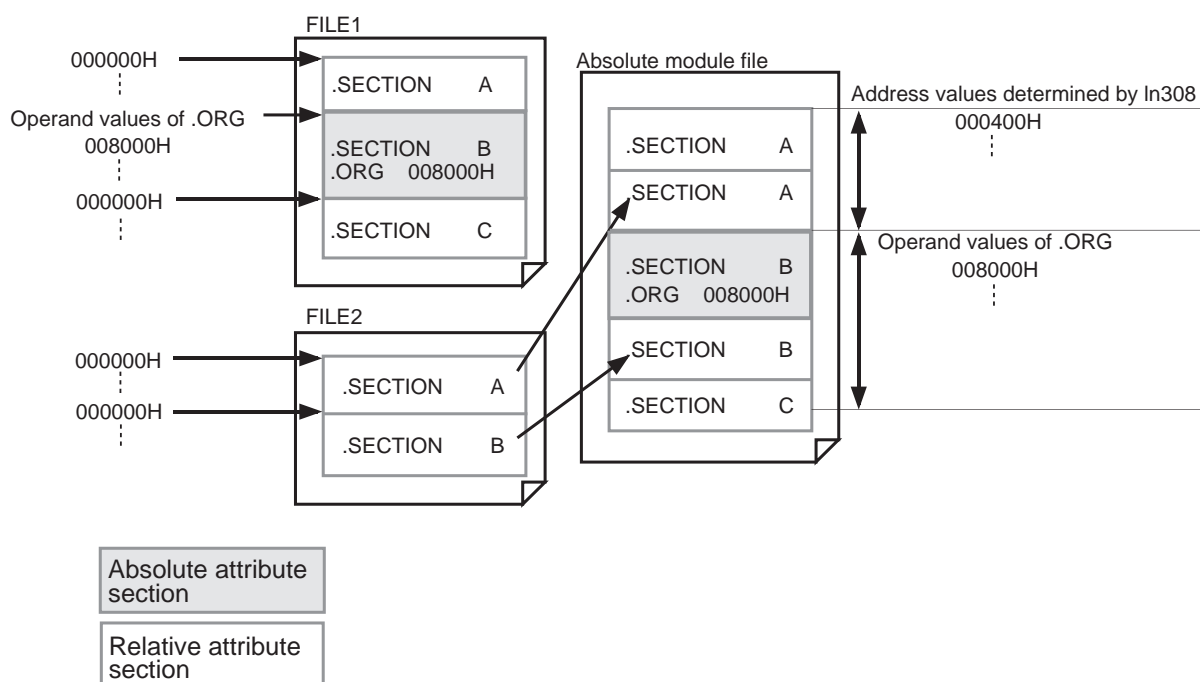


Figure 3.2.2 Example of address control

Reading Include File into Source Program

The AS308 system allows the user to read an include file into any desired line of the source program. This helps to increase the program readability.

Reading include file into source program

Write the file name to be read into the source program in the operand of directive command ".INCLUDE". All contents of the include file are read into the source program at the position of this line.

Example:

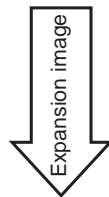
```
.INCLUDE      initial.inc
```

Source file (sample.a308)

```
work: .SECTION      memory,DATA
flags: .BLKB  10
      .BLKW  1
      .SECTION      init
      .INCLUDE      initial.inc
      .SECTION      program,CODE
main:
      .END
```

Include file (initial.inc)

```
loop:
      MOV.W      #10,A0
      MOV.B      #0,work[A0]
      INC.W      A0
      JNZ        loop
      MOV.W      #0,flags
```



After program is assembled

```
000000      work:      .SECTION      memory,DATA
00000A      flags:     .BLKB  10
000000      .SECTION      init
000000      .INCLUDE      initial
000000      loop:      MOV.W      #10,A0
000002      MOV.B      #0,work[A0]
000006      INC.W      A0
000007      JNZ        loop
000009      MOV.W      #0,flags
000000      .SECTION      program,CODE
main:
      .END
```

Addresses output by as308

Figure 3.2.3 Reading include file into source program

Global and Local Address Control

The following explains how the values of labels, symbols, and bit symbols are controlled by the AS308 system.

The AS308 system classifies labels, symbols, and bit symbols between global and local and between relocatable and absolute as it handles them. These classifications are defined below.

- **Global**

The labels and symbols specified with directive command ".GLB" are handled as global labels and global symbols, respectively.

The bit symbols specified with directive command ".BTGLB" are handle as global bit symbols.

If a name defined in the source file is specified as global, it is made referencible from an external file.

If a name not defined in the source file is specified as global, it is made an external reference label, symbol, or bit symbol that references a name defined in an external file.

- **Local**

All names are handled as local unless they are specified with directive command ".GLB" or ".BTGLB".

Local names can be referenced in only the same file where they are defined.

Local names are such that the same label name can be used in other files.

- **Relocatable**

The values of local labels, symbols, and bit symbols within relative sections are made relocatable.

The values of externally referenced global labels, symbols, and bit symbols are made relocatable.

- **Absolute**

The values of local labels, symbols, and bit symbols defined in an absolute attribute section are made absolute.

The labels, symbols, and bit symbols handled as absolute have their values determined by as308.

The values of all other labels, symbols, and bit symbols are determined by In308^(Note) when linking programs.

Figure 3.2.4 shows the relationship of various types of labels.

Note: A warning is output if linking resulted in any of these values exceeding the assembler-determined range in which branch instructions or addressing modes can be specified.

file1.a30

```

.GLB ver,sub1,port
.SECTION device
.ORG 400H
port: .BLKW 1
.SECTION program
.ORG 8000H
main:
JSR sub1
.SECTION str,ROMDATA
ver: .BYTE "program version 1"
.END

```

Declaration of label as global (essential)

Absolute labels in file1

port :Global; it can be referenced from external file.

main :Local

Relocatable labels in file1

ver :Global; it can be referenced from external file.

sub1 :Global; it references external file.

file2.a30

```

.GLB ver,sub1,port
.SECTION program
.ORG 0C000H
sub1:
LDM.W #0,A0
loop_s1:
LDM.B ver[A0],port
INC.W A0
CMP.B ver[A0],0
JNZ loop_s1
RTS
.END

```

Declaration of label as global (essential)

Absolute labels in file2

sub1 :Global; it can be referenced from external file.

loop_s1 :Local

Relocatable labels in file2

ver :Global; it references external file.

port :Global; it references external file.

Figure 3.2.4 Relationship of labels

3.2.3 Directive Commands

In addition to the M16C/80 series machine language instructions, the directive commands of the AS308 system can be used in the source program. Following types of directive commands are available. This section explains how to use each type of directive command.

(1)Address control command

To direct address determination when assembling the program.

(2)Assemble control directive command

To direct execution of AS308.

(3)Link control directive command

To define information for controlling address relocation.

(4)List control directive command

To control the format of list files generated by AS308.

(5)Branch optimization control directive command

To direct selection of the optimum branch instruction to AS308.

(6)Conditional assemble control directive command

To choose a block for which code is generated according to preset conditions when assembling the program.

(7)Extended function directive command

To exercise other control than those described above.

(8)Directive commands output by cross tools

The directive commands output by the M16C/80-series tool software cannot be written in source programs by the user. If this precaution is neglected, program operation cannot be guaranteed.

Address Control

Command	Function	Usage and Description Example
.ORG	Declares an address.	Write this command immediately after directive command ".SECTION". Unless this command is found immediately after the section directive command, the section is not made a relative attribute section. This command cannot be written in relative attribute sections. <pre> .ORG 0F0000H .ORG offset .ORG 0F0000H + offset </pre>
.BLKB	Allocates a RAM area in units of 1 byte.	Write the number of areas to be allocated in the DATA section. When defining a label name, always be sure to add a colon (:). Example: <pre> .BLKB 1 .BLKW number .BLKA number+1 label: .BLKL 1 label: .BLKF number label: .BLKD number+1 </pre>
.BLKW	Allocates a RAM area in units of 2 bytes.	
.BLKA	Allocates a RAM area in units of 3 bytes.	
.BLKL	Allocates a RAM area in units of 4 bytes.	
.BLKF	Allocates a RAM area for floating-point numbers in units of 4 bytes.	
.BLKD	Allocates a RAM area in units of 8 bytes.	
.BYTE	Stores data in the ROM area in length of 1 byte.	When writing multiple operands, separate them with a comma (,). When defining a label, always be sure to add a colon (:). For .FLOAT and .DOUBLE, write a floating-point number in the operand. Example: <pre> .SECTION value,ROMDATA .BYTE 1 .BYTE 1,2,3,4,5 .WORD "da","ta" .ADDR symbol .LWORD symbol+1 .FLOAT 5E2 constant .DOUBLE 5e2 </pre>
.WORD	Stores data in the ROM area in length of 2 bytes.	
.ADDR	Stores data in the ROM area in length of 3 bytes.	
.LWORD	Stores data in the ROM area in length of 4 bytes.	
.FLOAT	Stores a floating-point number in the ROM area in length of 4 bytes.	
.DOUBLE	Stores a floating-point number in the ROM area in length of 8 bytes.	
.ALIGN	Corrects odd addresses to even addresses.	This command can be written in the relative or absolute attribute section where address correction is specified when defining a section. Example: <pre> .SECTION work,DATA,ALIGN ram1: .BLKB 1 .ALIGN ram2: .BLKW 2 .SECTION const,ROMDATA .ORG 0F000H data1: .BYTE 12H .ALIGN data 2: .WORD symbol .END </pre>

Assemble Control

Command	Function	Usage and Description Example
.EQU	Defines a symbol.	<p>Forward referenced symbol names cannot be written. A symbol or expression can be written in the operand. Symbols and bit symbols can be specified as global.</p> <p>Example:</p> <pre>symbol .EQU 1 symbol1.EQU symbol+symbol bit0 .BTEQU 0,0 bit1 .BTEQU 1,symbol1</pre>
.BTEQU	Defines a bit symbol.	
.END	Declares the end of the assemble source.	<p>Write at least one instance of this command in one assembly source file. For lines following this directive command, as308 does not perform code generation or any other processing.</p> <p>Example:</p> <pre>.END</pre>
.SB	Assumes an SB register value.	<p>Always be sure to set each register before choosing the desired addressing mode. Since register values are not set in the actual register, write an instruction to set the register value immediately before or after this directive command.</p> <p>Example:</p> <pre>.SB 400H LDC #400H,SB .SBSYM sym1,sym2 .FB 500H LCD #580H,FB .FBSYM sym3,sym4</pre>
.SBSYM	Chooses SB relative addressing.	
.SBBIT	Chooses bit instruction SB relative addressing.	
.FB	Assumes an FB register value.	
.FBSYM	Chooses FB relative addressing.	
.INCLUDE	Reads a file into a specified position.	<p>Always be sure to write the extension for the file name in the operand. Directive command ".FILE" or a character string including "@" can be written in the operand.</p> <p>Example:</p> <pre>.INCLUDE initial.a30 .INCLUDE ..FILE@.inc</pre>

Link Control

Command	Function	Usage and Description Example
.SECTION	Defines a section name.	<p>When specifying section type and ALIGN simultaneously, separate them with a comma. The section type that can be written here is CODE, ROMDATA, or DATA. If section type is omitted, CODE is assumed.</p> <p>Example:</p> <pre> .SECTION program, CODE NOP .SECTION ram, DATA .BLK 10 .SECTION dname, ROMDATA .BYTE "abcd" .END </pre>
.GLB	Specifies a global label.	<p>When writing multiple symbol names in operand, separate them with a comma (.).</p> <p>Example:</p> <pre> .GLB name1, name2, mane3 .BTGLB flag4 .SECTION program MOV.W #0, name1 BCLR flag4 </pre>
.BTGLB	Specifies a global bit symbol.	
.VER	Outputs a specified character string to a map file as version information.	<p>Write operands within one line. This command can be written only once in one assembly source file.</p> <p>Example:</p> <pre> .VER 'strings' .VER "strings" </pre>

List Control

Command	Function	Usage and Description Example
.LIST	Controls line data output to a list file.	Write 'OFF' in the operand to stop line output or 'ON' to start line output. If this specification is omitted, all lines are output to the list file. Example: <pre>.LIST OFF MOV.B #0,R0L MOV.B #0,R0L MOV.B #0,R0L .LIST ON</pre>
.PAGE	Breaks page at a specified position in a list file.	Enclose the operand with single (') or double (") quotations when writing it. The operand can be omitted. Example: <pre>.PAGE .PAGE "strings" .PAGE 'strings'</pre>
.FORM	Specifies a number of columns and number of lines in one page of a list file.	This command can be written a number of times in one assembly source file. Symbols can be used to specify the number of columns or lines. Forward referenced symbols cannot be used, however. If this specification is omitted, the list file is output with 140 columns and 66 lines per page. Example: <pre>.FORM 20,80 .FORM 60 .FORM ,100 .FORM line,culmn</pre>

Branch Instruction Optimization Control

Command	Function	Usage and Description Example
.OPTJ	Controls optimization of branch instruction and subroutine call.	Various items can be written in the operand here, such as those for optimum control of a branch instruction and selection of an unconditional branch instruction or subroutine call instruction to be excluded from optimization. These items can be specified in any order and can be omitted. If omitted, the initial value or previously specified content is assumed for the jump distance. Example: Following combinations of operands can be written. <pre>.OPTJ OFF .OPTJ ON .OPTJ ON,JMPW .OPTJ ON,JMPW,JSRW .OPTJ ON,JMPW,JSRA .OPTJ ON,JMPA .OPTJ ON,JMPA,JSRW .OPTJ ON,JMPA,JSRA .OPTJ ON,JMRW .OPTJ ON,JMRA</pre>

Extended Function Directive Commands

Command	Function	Usage and Description Example
.ASSERT	Outputs a specified character string to a file or standard error output device.	<p>When outputting a character string enclosed with double quotations to a file, specify the file name following ">" or ">>". The bracket ">" creates a new file, so a message is output to it. If a file of the same name exists, a message is overwritten in it. The bracket ">>" outputs a message along with the contents of the file. If the specified file does not exist, it creates a new file. Directive command "..FILE" can be written in the file name.</p> <p>Example:</p> <pre>.ASSERT "string" > sample.dat .ASSERT "string" >> sample.dat .ASSERT "string" > ..FILE</pre>
?	Specifies and references a temporary label.	<p>Write "?:" in the line to be defined as a temporary label. To reference a temporary label that is defined immediately before, write "?-" in the instruction operand. To reference a temporary label that is defined immediately after, write "?+" in the instruction operand.</p> <p>Example:</p> <pre>?: JMP ?+ JMP ?-</pre>
..FILE	Indicates source file name information.	<p>This command can be written in the operand of directive command ".ASSERT" or ".INCLUDE". If command option "-F" is specified, "..FILE" is fixed to the source file name that is specified in the command line. If the option is omitted, the indicated source file name is the file name where "..FILE" is written.</p> <p>Example:</p> <pre>.ASSERT "sample" > ..FILE .INCLUDE ..FILE@.inc .ASSERT "sample" > ..FILE@.mes</pre>
@	Concatenates character strings before and after @.	<p>This command can be written a number of times in one line. If the concatenated character strings are going to be used as a name, do not enter a space or tab before and after this command.</p> <p>Example:</p> <pre>.ASSERT "sample" > ..FILE@.dat</pre> <p>Following macro definition is also possible:</p> <pre>mov_nibble .MACRO p1,src,p2,dest MOV@p1@p2 src,dest .ENDM</pre>

Conditional Assemble Directive Commands

Command	Function	Usage and Description Example
.IF	Indicates the beginning of conditional assemble.	<p>Always be sure to write a conditional expression in the operand.</p> <p>Example:</p> <pre>.IF TYPE==0 .BYTE "Proto Type Mode" .ELSE TYPE>0 .BYTE "Mass Production Mode" .ELSE .BYTE "Debug Mode" .ENDIF</pre> <p>Rules for writing conditional expression: The assembler does not check whether the operation has resulted in an overflow or underflow. Symbols cannot be forward referenced (i.e., symbols defined after this directive command are not referenced). If a forward referenced or undefined symbol is written, the assembler assumes value 0 for the symbol as it evaluates the expression.</p> <p>Typical description of conditional expression:</p> <pre>sym < 1 sym < 1 sym+2 < data1 sym+2 < data1+2 'smp1' ==name</pre>
.ELIF	Indicates condition for conditional assemble.	<p>Always be sure to write a conditional expression in the operand. This directive command can be written a number of times in one conditional assemble block.</p> <p>Example:</p> <p>Same as described above</p>
.ELSE	Indicates the beginning of a block to be assembled when condition is false.	<p>This directive command can be written more than once in the conditional assemble block. This command does not have an operand.</p> <p>Example:</p> <p>Same as described above</p>
.ENDIF	Indicates the end of conditional assemble.	<p>This directive command must be written at least once in the conditional assemble block. This command does not have an operand. Example:</p> <p>Same as described above</p>

3.2.4 Macro Functions

This section explains the macro functions that can be used in AS308. The following shows the macro functions available with AS308:

- **Macro function**

A macro function can be used by defining it with macro directive commands ".MACRO" to ".ENDM" and calling the defined macro.

- **Repeat macro function**

A repeat macro function can be used by writing macro directive commands ".MREPEAT" to ".ENDM".

Figure 3.2.5 shows the relationship between macro definition and macro call.

Macro Definition

To define a macro, use macro directive command ".MACRO" and define a set of instructions consisting of more than one line in one macro name. Use ".ENDM" to indicate the end of definition. The lines enclosed between ".MACRO" and ".ENDM" are called the macro body.

All instructions that can be written in the source program but a bit symbol can be used in the macro body. Macros can be nested in up to 65,535 levels including macro definitions and macro calls. Macro names and macro arguments are case-sensitive, so they are discriminated between uppercase and lowercase letters.

Macro Call

The contents of the macro body defined as a macro can be called into a line by writing the macro name defined with directive command ".MACRO" in that line. Macro names cannot be referenced externally. When calling the same macro from multiple files, define a macro in an include file and include that file to call the macro.

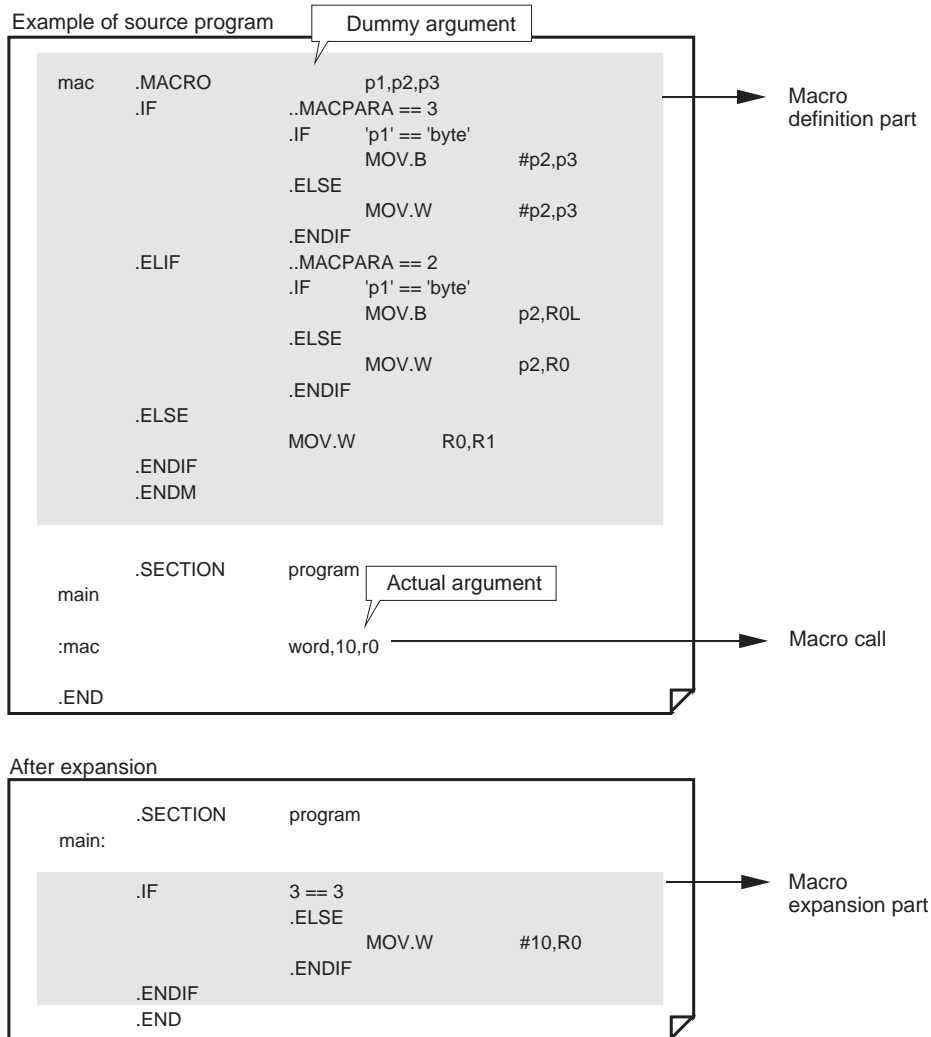


Figure 3.2.5 Example 1 of macro definition and macro call

Macro Local

Macro local labels declared with directive command ".LOCAL" can be used in only the macro definition. Labels declared to be macro local are such that the same label can be written anywhere outside the macro. Figure 3.2.6 shows a description example. In this example, m1 is the macro local label.

```

name     .MACRO      source,dest,top
      .LOCLA      m1
m1:
      nop
      jmp      m1
      .ENDM
  
```

Figure 3.2.6 Example 2 of macro definition and macro call

Repeat Macro Function

The macro body enclosed with macro directive commands ".MREPEAT" and ".ENDM" is expanded into a specified line repeatedly as many times as specified. Macro call of a repeat macro is not available.

Figure 3.2.7 shows the relationship between macro definition and macro call of a repeat macro.

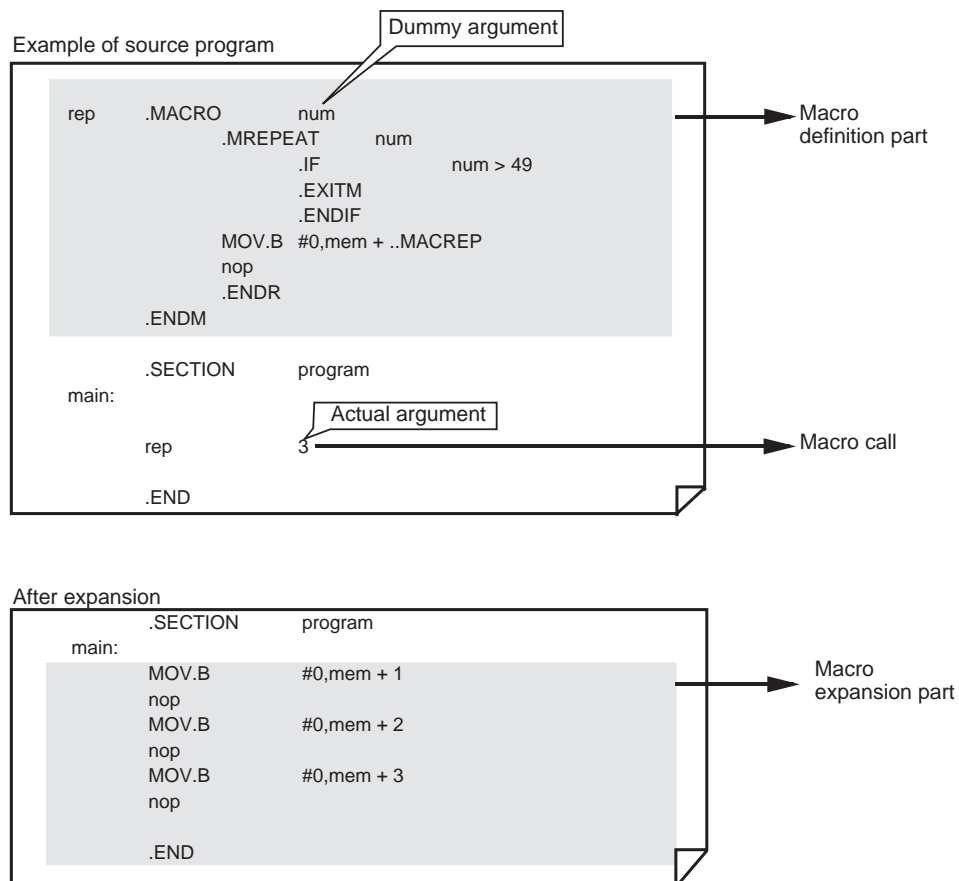


Figure 3.2.7 Example 3 of macro definition and macro call

Macro Directive Commands

There are following types of macro commands available with AS308:

- **Macro directive commands**

These commands indicate the beginning, end, or suspension of a macro body and declare a local label in the macro.

- **Macro symbols**

These symbols are written as terms of an expression in macro description.

- **Character string functions**

These functions show information on a character string.

Macro Directive Commands

Command	Function	Usage and Description Example
.MACRO	Defines a macro name and indicates the beginning of macro definition.	Always be sure to write a conditional expression in the operand. Up to 80 dummy arguments can be written. Do not enclose a dummy argument with double quotations. <Description format> Macro definition (macro name) .MACRO [(dummy argument) [(dummy argument)...]] Macro call (macro name) [(actual argument)][(actual argument)...]] <Description example> Refer to Figure 3.2.5.
.ENDM	Indicates the end of macro definition.	Write this command in relation to ".MACRO". <Description example> Refer to Figure 3.2.5.
.LOCAL	Declares that the label shown in the operand is a macro local label.	Write this command within the macro body. Multiple labels can be written by separating operands with a comma. The maximum number of labels that can be written in this way is 100. <Description example> Refer to Figure 3.2.6.
.EXITM	Forcibly terminates expansion of a macro body.	Write this command within the body of macro definition. <Description example> Refer to Figure 3.2.7.
.MREPEAT	Indicates the beginning of repeat macro definition.	The maximum number of repetitions is 65,535. <Description example> Refer to Figure 3.2.7.
.ENDR	Indicates the end of repeat macro definition.	Write this command in relation to ".MREPEAT". <Description example> Refer to Figure 3.2.7.

Macro Symbol

Command	Function	Usage and Description Example
..MACPARA	Indicates the number of actual arguments given when calling a macro.	This symbol can be written in the body of macro definition as a term of an expression. If written outside the macro body, value 0 is assumed. <Description example> Refer to Figure 3.2.5.
..MACREP	Indicates the number of times a repeat macro is expanded.	This symbol can be written in the body of macro definition as a term of an expression. It can also be written as an operand of conditional assemble. The value increments from 1 to 2, 3, and so on each time the macro is repeated. If written outside the macro body, value 0 is assumed. <Description example> Refer to Figure 3.2.7.

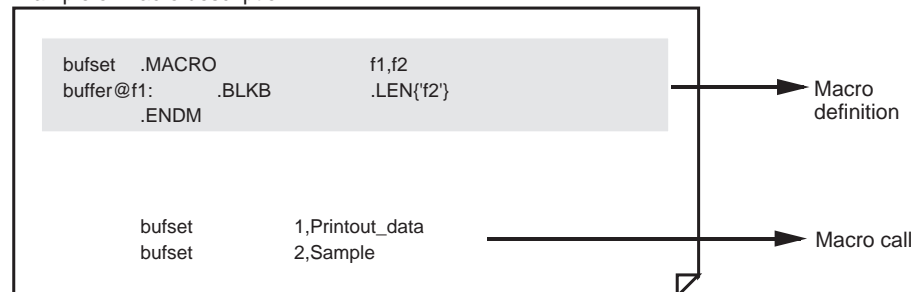
Character String Function

Command	Function	Usage and Description Example
.LEN	Indicates the length of a character string written in operand.	Always be sure to enclose the operand with brackets { } and the character string with quotations. Character strings can be written using 7-bit ASCII code characters. This function can be written as a term of an expression. <Description format> .LEN { "(string)" } .LEN { '(string)' } <Description example> Refer to Figure 3.2.8.
.INSTR	Indicates the start position of a search character string in character strings specified in operand.	Always be sure to enclose the operand with brackets { } and the character string with quotations. Character strings can be written using 7-bit ASCII code characters. If the search start position = 1, it means the beginning of a character string. <Description format> .INSTR { "(string)", "(search character string)", (search start position) } .INSTR { '(string)', '(search character string)', (search start position) } <Description example> Refer to Figure 3.2.9.
.SUBSTR	Extracts a specified number of characters from the character string position specified in operand.	Always be sure to enclose the operand with brackets { } and the character string with quotations. Character strings can be written using 7-bit ASCII code characters. If the extraction start position = 1, it means the beginning of a character string. <Description format> .SUBSTR { "(string)", (start position), (number of characters) } .SUBSTR { '(string)', (start position), (number of characters) } <Description example> Refer to Figure 3.2.10.

Example of .LEN Statement

In the example of Figure 3.2.8, the length of a specified character string is "13" for "Printout_data" and "6" for "Sample".

Example of macro description



Macro expansion

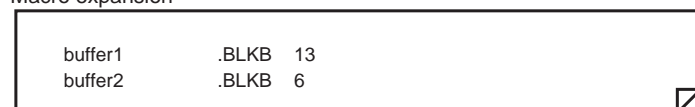
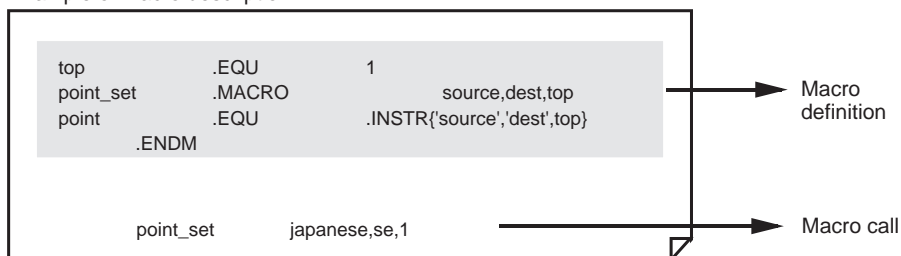


Figure 3.2.8 Example of .LEN statement

Example of .INSTR Statement

In the example of Figure 3.2.9, the position (7) of character string "se" from the beginning x (top) of a specified character string (japanese) is extracted.

Example of macro description



Macro expansion

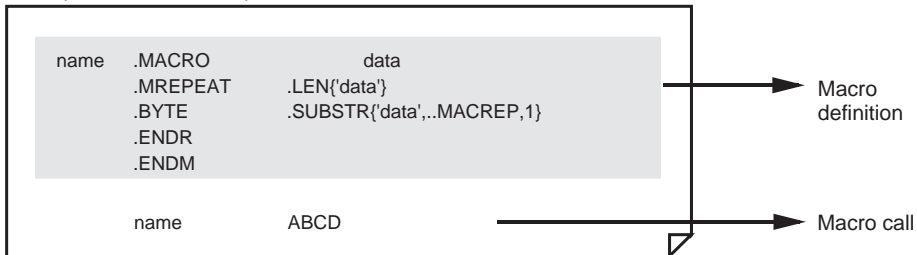


Figure 3.2.9 Example of .INSTR statement

Example of .SUBSTR Statement

In the example of Figure 3.2.10, the length of a character string given as the macro's actual argument is given to the operand of ".MREPEAT". Each time the ".BYTE" line is executed, ".MACREP" is increased from 1 to 2, 3, 4, and so on. Consequently, characters are passed one character at a time from the character string given as the actual macro argument to the operand of ".BYTE" sequentially beginning with the first character.

Example of macro description



Macro expansion

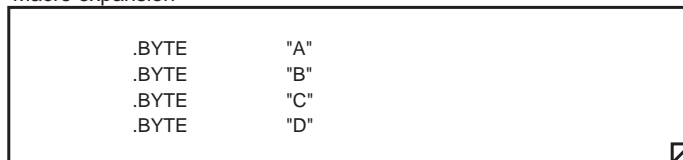


Figure 3.2.10 Example of .SUBSTR statement

3.2.5 Differences with M16C/60

AS308 (M16C/80) has new addressing modes, instruction sets, and assemble options that have been added or changed from AS30 (M16C/60). But there are some addressing modes, instruction sets, and assemble options that are removed and not included in AS308. This section describes those that have been removed or changed.

Changed addressing modes

As the memory space in M16C/80 has been expanded from 1M to 16M, the address ranges that can be accessed in all addressing modes, such as general instruction addressing and bit instruction addressing, have been expanded. Namely, locations following address 100000H can now be accessed.

Therefore, some addressing modes have become unusable in M16C/80, as described below.

(1) Following modes of specific instruction addressing have been removed:

- (a) 20-bit absolute
- (b) Address register relative with 20-bit displacement attached
- (c) 32-bit register direct
- (d) 32-bit address register indirect

* In M16C/80, general instruction addressing can be used for all of the above.

(2) Following modes of bit instruction addressing have been removed:

(a) Register direct

For register bit manipulation in M16C/80, bits 0 to 7 only can be specified.

Register direct in M16C/60

(bits 0 to 15)

bit, R0

bit, R1

bit, R2

bit, R3

bit, A0

bit, A1



Register direct in M16C/80

(bits 0 to 7)

bit, R0L

bit, R0H

bit, R1L

bit, R1H

bit, A0(Only 8 low-order bits can be specified)

bit, A1(Only 8 low-order bits can be specified)

Removed instruction sets

Following instructions have been removed and are not included in M16C/80:

- (1)LDE instruction
- (2)STE instruction
- (3)LDINTB (LDC macro) instruction

* Because the memory space in M16C/80 has been expanded to 16M and because locations following address 100000H can also be accessed by general instruction addressing, LDE and STE instructions have been removed.

Compatibility with M16C/60 instructions (1)

In some instructions of the M16C/80, src and dest that can be selected by each instruction (i.e., usable operands) are different from M16C/60.

AS308 supports "-mode60" as a command option necessary to assemble programs developed by AS30 (M16C/60 series). The following shows how instructions are processed by AS308 when this option is added.

- (1)The format specifier written in MOV, CMP, ADD, SUB, AND, OR, NOT, PUSH, or POP instruction is ignored.
- (2)The addressing mode specifier of JMPL and JSRL instructions are ignored.
- (3)When adding to the stack pointer (SP) in ADD instruction, the size specifier ".L" is assumed.
- (4)The LDINTB instruction is replaced with the LDC instruction when processing the instruction. Refer to Table 3.2.9.
- (5)The operands of STZ, STNZ, and STZX instructions are processed in byte size.
- (6)The LDE and STE instructions are replaced with the MOV instruction when processing the instruction. Refer to Table 3.2.9.
- (7)The 1-bit manipulate instruction is replaced with the corresponding AS308 (M16C/80) instruction when processing the instruction. Refer to Table 3.2.9
- (8)The bit manipulate instructions BCLR, BAND, BOR, BXOR, BNOT, BNAND, BNOR, BNXOR, BTST, BNTST, BTSTC, BTSTS, and BMcmd also are replaced in the same way as for the BSET instruction shown in the replacement instruction list.

List of instructions replaced by the "-mode60" option

Table 3.2.9 Replacement Instruction List

AS30 source description format		Results when replaced in AS308	
LDINTB	#imm20	LDC	#imm24,INTB
LDE.B/W	dsp:20, dest	MOV.B/W	abs, dest
LDE.B/W	dsp:20[A0], dest	MOV.B/W	dsp[A0], dest
STE.B/W	src, abs:20	MOV.B/W	src, abs
STE.B/W	src ,dsp:20[A0]	MOV.B/W	src, desp[A0]
BSET:G	bit, R0	BSET BSET	bit, R0L bit, R0H
BSET:G	bit, R1	BSET BSET	bit, R1L bit, R1H
BSET:G	bit, A0	Can be assembled for bit positions 0 to 7	
BSET:G	bit, A1	Can be assembled for bit positions 0 to 7	
BSET:G	bit, [A0]	BITINDEX.B BSET	[A0] 0, 0
BSET:G	bit, [A1]	BITINDEX.B BSET	[A1] 0, 0
BSET:G	bit, base:8[A0]	BITINDEX.B BSET	[A0] 0, base
BSET:G	bit, base:16[A0]	BITINDEX.B BSET	[A0] 0, base
BSET:G	bit, base:8[A1]	BITINDEX.B BSET	[A1] 0, base
BSET:G	bit, base:16[A1]	BITINDEX.B BSET	[A1] 0, base
BSET:G BSET:G BSET:G	bit, base:8[SB] bit, base:11[SB] bit, base:16[SB]	BSET	bit, base[SB]
BSET:G	bit, base:8[FB]	BSET	bit, base[FB]
BSET:G	bit, base:16	BSET	bit, base

Compatibility with M16C/60 instructions (2)

The following instructions cannot be replaced with M16C/80 equivalents even by using the command option "-mode60." For these instructions, compatibility can be maintained by changing the source program directly.

- (1)MOVA src, R0
- (2)MOVA src, R1
- (3)MOVA src, R2
- (4)MOVA src, R3
- src = dsp[A0],dsp[A1],dsp[SB],dsp[FB],abs16
- (5)JMPL.A A1A0
- (6)JSRI.A A1A0
- (7)PUSHC INTBL
- (8)PUSHC INTBH
- (9)POPC INTBL
- (10)POPC INTBH
- (11)MUL.W generic, A0
- (12)MULU.W generic, A0
- generic = R0, R1, R2, R3, A0, A1, [A0], [A1],
- dsp[SB], dsp[FB], dsp[A0], dsp[A1], abs16
- (13)LDC
- (14)STC
- (15)LDE.B/W [A1A0], generic
- (16)STE.B/W generic, [A1A0]
- generic = R0L/R0, R0H/R1, R1L/R2, R1H/R3, A0, A1, [A0], [A1],
- dsp[SB], dsp[FB], dsp[A0], dsp[A1], abs16
- (17)BSET:G bit, R2
- (18)BSET:G bit, R3
- (19)Bit manipulate instructions BCLR, BAND, BOR, BXOR, BNOT, BNAND, BNOR, BNxor,
 BTST, BNTST, BTSTC, BTSTS, and BMcnd which are written in the same way as "BSET:G
 bit,R2/R3" instruction.

Removed assemble options

AS308 has had the following options removed.

- (1)-M60, -M61, -M62, -M62E (options for identification of M16C groups)
- (2)-A (option for operand evaluation in mnemonic)
- (3)-P (option for structured description instructions)

Option for structured description

Although AS308 does not support the structured description of instructions, it supports "-mode60p" as an option necessary to assemble programs developed by AS30 (M16C/60) using structured description.

- (1)-mode60p

After starting the structured description preprocessor (pre30) that accompanies the structured description of AS30, the command option "-mode60" is processed.^(note)

Note: Because AS308 does not support structured description, not all structured sources can be processed by AS308. Therefore, structured description that expands into nonexistent mnemonics of AS308 or mnemonics that behave differently in AS308 cannot be handled by adding this option. In such a case, the assembly source must be changed by the user.

Chapter 4

Programming Style

- 4.1 Hardware Definition
- 4.2 Initial Setting of CPU
- 4.3 Setting when using Interrupts
- 4.4 Dividing Source File
- 4.5 A Little Tips...(Programing Technique)
- 4.6 Standard processing program

4.1 Hardware Definition

This section explains how to define an SFR area and create an include file, how to allocate RAM data and ROM data areas, and how to define a section.

4.1.1 Defining SFR Area

It should prove convenient to create the SFR area's definition part in an include file. There are two methods for defining the SFR area as described below.

Definition by .EQU

Figure 4.1.1 shows an example for defining the SFR area by using directive command ".EQU".

```

;-----
; M30800 SFR Definition File
;-----
PM0      .EQU      0004H ; Processor mode register 0
PM1      .EQU      0005H ; Processor mode register 1
CM0      .EQU      0006H ; System clock control register 0
CM1      .EQU      0007H ; System clock control register 1
WCR      .EQU      0008H ; Wait control register
AIER     .EQU      0009H ; Address match interrupt enable register
PRCR     .EQU      000AH ; Protect register
DS       .EQU      000BH ; External data bus widthcontrol register
MCD      .EQU      000CH ; Main clock division register
;
WDTS     .EQU      000EH ; Watchdog timer start register
WDC      .EQU      000FH ; Watchdog timer control register
RMAD0    .EQU      0010H ; Address match instruction register 0
RMAD1    .EQU      0014H ; Address match instruction register 1
RMAD2    .EQU      0018H ; Address match instruction register 2
RMAD3    .EQU      001CH ; Address match instruction register 3
;
EIAD     .EQU      0020H ; Emulator interrupt vector table register
EITD     .EQU      0023H ; Emulator interrupt detect register
EPRR     .EQU      0024H ; Emulator protect register
ROA      .EQU      0030H ; ROM areaset register

```

Define the address at which processor mode register 0 is placed. In the following lines, define the addresses of other registers.

Define the start address of a register that consists of more than 2 bytes.

Figure 4.1.1 Example of SFR area definition by ".EQU"

Definition by .BLKB

Figure 4.1.2 shows an example for defining the SFR area by using directive command ".BLKB".

```

;-----
; M30800 SFR Definition File
;-----
; Declare a section name.
SECTION SFR_DATA
; Specify an absolute address
; according to the address at which
; processor mode register 0 is placed.
.ORG 000004H

PM0 .BLKB 1 ; Processor mode register 0
PM1 .BLKB 1 ; Processor mode register 1
CM0 .BLKB 1 ; System clock control register 0
CM1 .BLKB 1 ; System clock control register 1
WCR .BLKB 1 ; Wait control register
AIER .BLKB 1 ; Address match interrupt enable register
PRCR .BLKB 1 ; Protect register
DS .BLKB 1 ; External data bus width control register
MCD .BLKB 1 ; Main clock division register

; Note that unless 0000EH is specified
; for the absolute address here, the area
; for the watchdog timer start register will
; be set at 0000BH, a location next to the
; protect register.
.ORG 00000EH

WDTS: .BLKB 1 ; Watchdog timer start register
WDC: .BLKB 1 ; Watchdog timer control register
RMAD0: .BLKA 1 ; Address match instruction register 0
      .BLKB 1
RMAD1: .BLKA 1 ; Address match instruction register 1
      .BLKB 1
RMAD2: .BLKA 1 ; Address match instruction register 2
      .BLKB 1
RMAD3: .BLKA 1 ; Address match instruction register 3
      .BLKB 1

; Allocate areas even for locations
; where nothing is placed.
.ORG 000020H
EIAD .BLKA 1 ; Emulator interrupt vector table register
EITD .BLKB 1 ; Emulator interrupt detect register
EPRR .BLKB 1 ; Emulator protect register

;
.ORG 000030H
ROA .BLKB 1 ; ROM area set register
DBA .BLKB 1 ; Debug monitor area set register
EXA .BLKB 1 ; Expansion area set register 0
EXA .BLKB 1 ; Expansion area set register 1

```

Figure 4.1.2 Example of SFR area definition by ".BLKB"

Creating Include File

When creating the source program in separate files, create an include file for SFR definition and other parts that are used by multiple files. Normally add an extension ".INC" for the include file.

Precautions on creating include file

(1) When using ".EQU" in include file

Directive command ".EQU" defines values for symbols. It can also be used to define addresses as in SFR definition. However, since this is not a command to allocate memory areas, make sure that the addresses defined with it will not overlap. The include file created using ".EQU" can be used in multiple files by reading it in.

(2) When using ".ORG" in include file

If an include file created using ".ORG" is read into multiple files, a link error will result. This is because the include file contains the absolute addresses specified by ".ORG". Consequently, the defined addresses overlap with each other.

(3) When using ".BLKB", ".BLKW", and ".BLKA" in include file

Directive commands ".BLKB", ".BLKW", and ".BLKA" are used to allocate memory areas. If an include file created using these directive commands is read into multiple files, areas will be allocated separately in each file. Although no error may occur when using symbols in the include file locally, care must be taken when using them globally because it could result in duplicate definitions.

If use of a common area in multiple files is desired, define the area-allocated part in a shared definition file and link it as one of the source files. Then define the symbol's global specification part in an include file.

Reading Include File into Source File

Use directive command ".INCLUDE" to read an include file into the source file. Specify the file name to be read in with a full name.

Example:

When reading an include file "M30800.INC" that contains a definition of the SFR area

```
.INCLUDE    M30800.INC
```

4.1.2 Allocating RAM Data Area

Use the following directive commands to allocate a RAM area:

.BLKB Allocates a 1-byte area (integer)
 .BLKW Allocates a 2-byte area (integer)
 .BLKA Allocates a 3-byte area (integer)
 .BLKL Allocates a 4-byte area (integer)
 .BLKF Allocates a 4-byte area (floating-point)
 .BLKD Allocates a 8-byte area (floating-point)

Example for Setting Up Work Area

Figure 4.1.3 shows an example for setting up a work area.

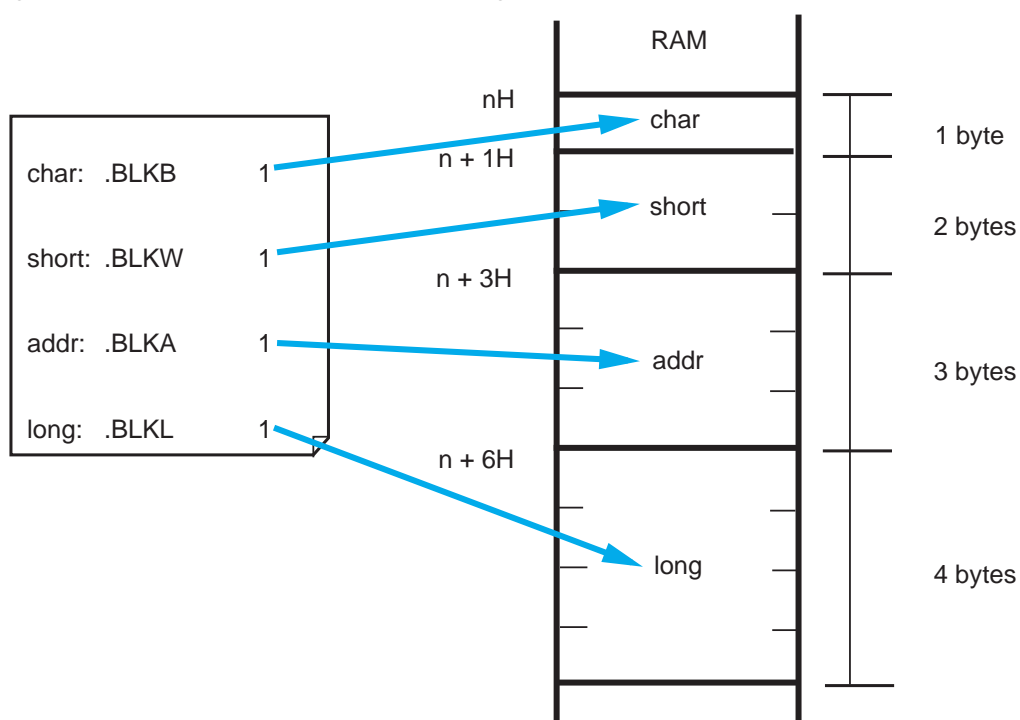


Figure 4.1.3 Example for setting up a work area

4.1.3 Allocating ROM Data Area

Use the directive commands listed below to set fixed data in ROM. For a description example, refer to Section 4.1.5, "Sample Program List 1 (Initial Setting 1)".

```
.BYTE ..... Sets 1-byte data (integer)
.WORD ..... Sets 2-byte data (integer)
.ADDR ..... Sets 3-byte data (integer)
.LWORD .... Sets 4-byte data (integer)
.FLOAT ..... Sets 4-byte data (floating-point)
.DOUBLE ... Sets 8-byte data (floating-point)
```

Retrieving Table Data

Figure 4.1.4 shows an example of a data table. Figure 4.1.5 shows a method for accessing this table by using address register relative addressing.

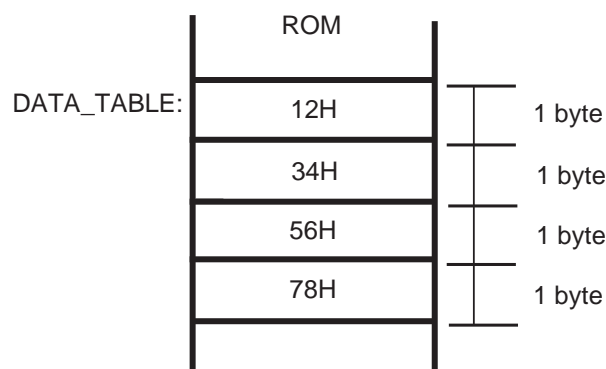


Figure 4.1.4 Example for setting a data table

```

      :
      :
MOV.W #1,A0
MOV.B DATA_TABLE[A0],R0L      ;Stores the data table's 2nd byte (34H) in R0L.
      :
      :
DATA_TABLE:
      .BYTE 12H,34H,56H,78H      ;Sets 1-byte data.
      :
      :
```

Figure 4.1.5 Example for retrieving data table

4.1.4 Defining a Section

Directive command ".SECTION" declares a section in which a program part from the line where this directive command is written to the next ".SECTION" is allocated.

Description Format of Section Definition

.SECTION section name [(section type), ALIGN]
Specification in [] can be omitted.

A range of statements from one directive command ".SECTION" to a position before the line where the next ".SECTION" or directive command ".END" is written is defined as a section. Any desired section name can be set. Furthermore, one of section types (DATA, CODE, or ROMDATA) can be set for each section. Note that the instructions which can be written in the section vary with this section type. For details, refer to AS308 User's Manual.

If ".ALIGN" is specified for a section, the linker (ln308) locates the beginning of the section at an even address.

Example for Setting Up Sections

Figure 4.1.6 shows an example for setting up each section.

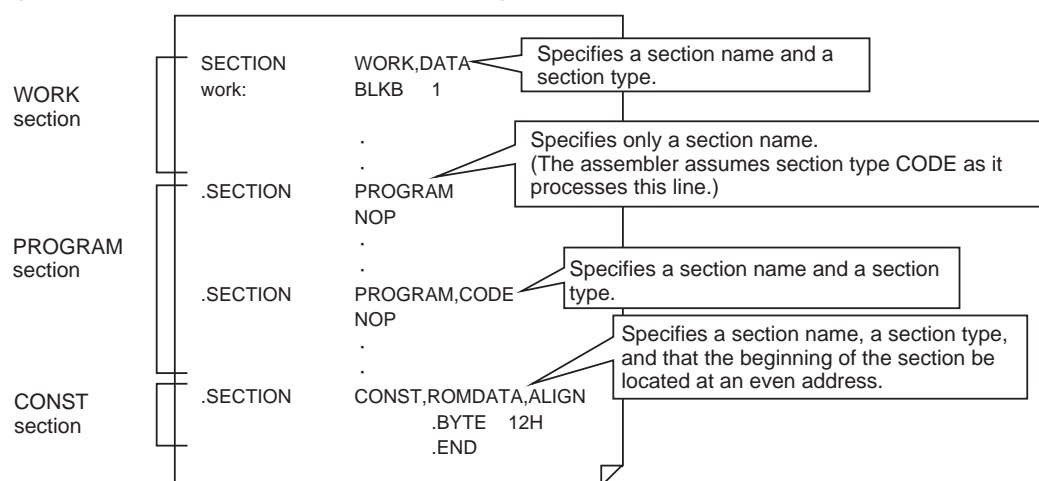


Figure 4.1.6 Example for setting up sections

Section Attributes

Each section is assigned an attribute when assembling the program. There are two attributes: relative and absolute.

(1) Relative attribute

- Location of each section can be specified when linking source files. (Relocatable)
- Addresses in the section are made relocatable values when assembling the program.
- The values of labels defined in this type of section become relocatable.

(2) Absolute attribute

- A section is assigned an absolute attribute and handled as such by specifying addresses with ".ORG" immediately after directive command ".SECTION".
- Addresses in the section are made relocatable values when assembling the program.
- The values of labels defined in this type of section become absolute.

```

***** Include *****
;
;
; .INCLUDE      M30800.INC          Reads include file into source file.
;
;
***** Symbol definition *****
;
;
RAM_TOP           .EQU    000400H           ;Start address of RAM
RAM_END           .EQU    002BFFH           ;End address of RAM
ROM_TOP           .EQU    0FE0000H         ;Start address of ROM
FIXED_VECT_TOP    .EQU    0FFFFDCH         ;Start address of fixed vector
SB_BASE           .EQU    000400H           ;Base address of SB relative addressing
FB_BASE           .EQU    000580H           ;Base address of FB relative addressing
ISTACK_SIZE       .EQU    300H             ;Interrupt stack area size
;
;
***** Allocation of work RAM area *****
;
;
        .SECTION      WORK,DATA           Matched to hardware RAM area.
        .ORG          RAM_TOP
;
WORKRAM_TOP:     Add ":" (colon) at the end of a label name.
char:            .BLKB      1              ;Allocates a 1-byte area.
short:           .BLKW     1              ;Allocates a 2-byte area.
addr:            .BLKA     1              ;Allocates a 3-byte area.
long:            .BLKL     1              ;Allocates a 4-byte area.
WORKRAM_END:
;
;
***** Definition of bit symbo *****
;
;
Do not add ":" (colon) for a bit symbol.
char_b0          .BTEQU    0,char         ; Bit 0 of char
short_b1         .BTEQU    1,short        ; Bit 1 of short
addr_b2          .BTEQU    2,addr         ; Bit 2 of addr
long_b3          .BTEQU    3,long         ; Bit 3 of long
;
;
***** Program area *****
;
=====Startup=====
;
;
        .SECTION      PROGRAM,CODE
        .ORG          ROM_TOP
;
START:
        LDC    #RAM_END+1,ISP             ; Sets initial value in stack pointer(ISP).
;
        MOV.B   #03H,PRCR                 ; Removes protection.
        MOV.W   #0183H,PM0                ; Sets processor mode register 0 and 1.
        MOV.W   #2008H,CM0               ; Sets system clock control register.
        MOV.B   #12H,MCD                  ; Sets main clock division register.
        MOV.B   #0,PRCR                   ; Protects all registers.
;

```

Must be matched to hardware and the contents selected in programming.

```

MOV.W    #0,PS0           ; Sets function select register A0 and A1.
MOV.B    #0,PS2           ; Sets function select register A2.
MOV.W    #0,PSL0          ; Sets function select register B0 and B1.
MOV.B    #0,PSL2          ; Sets function select register B2.
MOV.B    #0,PSC           ; Sets function select register C.
BSET     2,PRCR           ; Remote protection.
                        ; ( Write function select register A3 enabled)
MOV.B    #0,PS3           ; Sets function select register A3.
;
MOV.B    #03H,DS          ; Sets external data bus width control register.
MOV.B    #85H,WCR         ; Sets wait control register.
;
LDC      #80H,FLG         ; Sets initial value in flag register.(Sets U = 1)
LDC      #(RAM_END-ISTACK_SIZE)+1,SP ; Sets initial value in stack pointer(USP).
;
MOV.W    #0FFF0H,PUR2     ; Connects internal pull-up registers to ports P6 through P10.
;
MOV.B    #0FFH,03CBH      ; Sets initial value in SFR reserved area.
MOV.W    #0FFFFH,03CEH
MOV.W    #0FFFEH,03D2H
;
Declaration to the assembler.
.SB      SB_BASE          ; Values declared to the assembler are matched.
.FB      FB_BASE          ; Declares SB register value to the assembler.
;
LDC      #SB_BASE,SB      ; Sets initial value in SB register.
LDC      #FB_BASE,FB      ; Sets initial value in FB register.
;
MOV.W    #0,R0
MOV.W    #(RAM_END+1 - RAM_TOP)/2,R3 ; Clears WORK_RAM area in zero.

MOV.W    #WORKRAM_TOP,A1
SSTR.W
;
===== Main program =====
MAIN:
MOV.B    DATA_TABLE[A0],R0L
MOV.W    #1234H,R1
BSET     char_b0
;
;
JMP      MAIN
;
===== Dummy interrupt program =====
dummy:
REIT
;

```



```

;=====Fixed data area=====
;
; .SECTION    CONSTANT,ROMDATA    ; Declares section name and section type.
; .ORG        XXXXXH              ; Declares start address.
;
DATA_TABLE:
; .BYTE       12H,34H,56H,78H      ; Sets 1-byte data.
; .WORD       1234H,5678H          ; Sets 2-byte data.
; .ADDR       123456H,789ABCH      ; Sets 3-byte data.
; .LWORD      12345678H,9ABCDEF0H  ; Sets 4-byte data.
DATA_TABLE_END:
;
; ***** Setting of fixed vector *****
;
; .SECTION    F_VECT,ROMDATA
; .ORG        FIXED_VECT_TOP
; .LWORD      dummy                ; Undefined instruction interrupt vector
; .LWORD      dummy                ; Overflow (INTO instruction) interrupt vector
; .LWORD      dummy                ; BRK instruction interrupt vector
; .LWORD      dummy                ; Address match interrupt vector
; .LWORD      dummy                ; Not used.
; .LWORD      dummy                ; Watchdog timer interrupt vector
; .LWORD      dummy                ; Not used.
; .LWORD      dummy                ; NMI interrupt vector
; .LWORD      START                ; Sets reset vector.
;
; .END

```

Must be matched to ROM area in hardware.

Set jump addresses sequentially beginning with the least significant address of the fixed vector.

Set jump addresses for unused interrupts in dummy processing (REIT instruction only) to prevent the program from running out of control when an unused interrupt is requested.

Set the program start address for the reset vector. Immediately after power-on or after a reset is deactivated, the program starts from the address written in this vector.

Figure 4.1.7 Description example 1 for initial setting

4.2 Initial Setting the CPU

Each register as well as RAM and other resources must be initial set immediately after power-on or after a reset. If the CPU internal registers remain un-set or there is unintended data left in memory before program execution, all this could cause the program to run out of control. Therefore, the internal resources must be initial set at the beginning of the program. This initial setting includes the following:

- (1) Declaration to the assembler
- (2) Initialization of the CPU internal registers, flags, and RAM area
- (3) Initialization of work area
- (4) Initialization of built-in peripheral functions such as port, timer, and interrupt

4.2.1 Setting CPU Internal Registers

After a reset, it is normally necessary to set up the registers related to the processor's operation mode, system clock, and port functions.

Setting the processor mode and system clock

The Processor Mode Registers 0/1, System Clock Control Registers 0/1, and Main Clock Divide Register are protected registers, so remove protection of these registers before you set them and re-protect the registers after you finished setting them. Figure 4.2.1 shows an example of how to set the registers.

```

;-----Setting the processor mode and system clock-----
;
MOV.B    #03H,PRCR        ; Remove protection.
MOV.W    #0183H,PM0        ; Sets processor mode registers 0 and 1.
MOV.W    #2008H,CM0        ; Sets system clock control registers 0 and 1.
MOV.B    #12H,MCD         ; Sets main clock divide register.
MOV.B    #0,PRCR          ; Protects all registers.
;

```

Figure 4.2.1 Example for setting the processor mode and system clock

Setting port functions

If a pin output function in M16C/80 is multiplexed between port output and peripheral function output or a single pin is assigned multiple peripheral function outputs, it is necessary to select the desired output function using Function Select Registers. Figure 4.2.2 shows an example of how to set the Function Select Register.

```

;-----Setting Function Select Registers-----
;
MOV.W    #0,PS0      ; Set Function Select Registers A0 and A1
MOV.B    #0,PS2      ; Set Function Select Register A2
MOV.W    #0,PSL0     ; Set Function Select Registers B0 and B1
MOV.B    #0,PSL2     ; Set Function Select Register A2
MOV.B    #0,PSC      ; Set Function Select Register C
BSET     2,PRCR       ; Remote protection
                        ; (Write to Pin Function Select Register A3 enabled)
MOV.B    #0,PS3; Set Pin Function Select Register A3
;

```

Figure 4.2.2 Example for setting function select registers

4.2.2 Setting Stack Pointer

When using a subroutine or interrupt, the return address, etc. are saved to the stack. Therefore, the stack pointer must be set before calling the subroutine or enabling the interrupt. For a setup example, refer to Section 4.2.7, "Sample Program List 2 (Initial Setting 2)".

4.2.3 Setting Base Registers (SB, FB)

The M16C/80 series has an addressing mode called "base register relative addressing" to allow for efficient data access. Since a relative address from an address that serves as the base is used for access in this mode, it is necessary to set the base address before this addressing mode can be used. For a setup example, refer to Section 4.2.7, "Sample Program List 2 (Initial Setting 2)".

4.2.4 Setting fixed interrupt vector (reset vector)

The M16C/80 series has two types of vectors available, a variable and a fixed vector. For details on how to set the fixed interrupt vectors including a reset vector, refer to Section 4.2.6, "Sample List 2 (Initial Settings 2)."

4.2.5 Setting internal peripheral functions

The following explains how to set the internal RAM, ports, timers, and DMA controller of the M16C/80 group. For details, refer to the functional description in the user's manual supplied with your microcomputer.

Initial Setting Work Areas

Normally clear the work areas to 0 by initial setting. If the initial value is not 0, set that initial value in each work area. Figure 4.2.3 shows an example for initial setting a work area.

```

;-----Clearing work RAM to 0 by string instruction-----
RAM_TOP    .EQU    0400H
RAM_END    .EQU    2BFFH
;
;      MOV.W      #0,R0
;      MOV.W      #(RAM_END + 1 - RAM_TOP) / 2,R3
;      MOV.W      #WORKRAM_TOP,A1
;      SSTR.W
;                                     ; Transfer a 0 from WORKRAM_TOP
;                                     ; two times for (RAM_END + 1 - RAM_TOP).
;
;-----Setting initial values in work RAM-----
;      MOV.B      #0FFH,char          ; Set one byte of data.
;
;      MOV.B      #0FFFFH,short       ; Set one word of data.
;
;      MOV.W      #0FFFFH,addr        ; Set three bytes of data.
;      MOV.B      #0FFH,addr + 2
;
;      MOV.L      #0FFFFFFFFH,long    ; Set one long word of data.

```

Figure 4.2.3 Example for initial setting a work area

Initial Setting Ports

It is when a port direction register is set for output that data is output from a port. To prevent indeterminate data from being output from ports, set the initial value in each output port before setting their direction register for output. Figure 4.2.4 shows an example for initial setting ports.

```

;-----Initial setting ports-----
;
;
MOV.W #0FFFFH,P6      ; Sets initial value in ports P6 and P7.
MOV.W #0FFFFH,PD6     ; Sets ports P6 and P7 for output.
MOV.B #3CH,P9         ; Sets initial value in ports P9.
;
;
MOV.B #04H,PRCR       ; Removes protect.(Note1)
MOV.W #0FFH,PD9       ; Sets ports P9 for output.(Note2)
;
;

```

Figure 4.2.4 Example for initial setting ports

Setting Timers

When using the M16C/80 series built-in peripheral functions such as a timer, initial set the related registers (in SFR area). Figure 4.2.5 shows an example for setting timer A0.

```

;-----Setting Timer A0-----
;
;
TA0R .BTEQU      3,TA0IC
TA0S .BTEQU      0,TABSR
;
;
MOV.B      #01000000B,TA0MR ; Set Timer A0 Mode Register.
;                               ; (Mode: timer mode; divide ratio: 1/8)
MOV.W      #2500 - 1,TA0    ; Set Timer A0 count value.
BCLR      TA0R              ; Clear Timer A0 interrupt request bit.
;
;
BSET      TA0S              ; Timer A0 starts counting.
;
;

```

Figure 4.2.5 Example for setting timer

Note 1: Because the Port P9 Direction Register is a protected register, set the Protect Register bit 2 to 1 to remove the protection before you set a value.

Note 2: The Port P9 Direction Register write enable bit (Protect Register bit 2) is reset to 0 by the next write instruction executed after being write-enabled. Therefore, to change a port for input or output, be sure to set the Port P9 Direction Register immediately after the instruction by which its write enable bit is set to 1. Also, make sure no interrupt or DMA transfer will occur during this time.

Setting the DMA controller

When using the DMAC, initial set the registers associated with it (CPU internal registers and SFR area). The DMAC-related registers are shown in Figure 4.2.6.

----- When using 1-2 DMAC channels (DMA0, DMA1) -----

DMAC-related registers

	b7	b0	DMD0	DMA mode register 0
			DMD1	DMA mode register 1
b15			DCT0	DMA0 transfer count register
			DCT1	DMA1 transfer count register
			DRC0	DMA0 transfer count reload register
			DRC1	DMA1 transfer count reload register
b23			DMA0	DMA0 memory address register
			DMA1	DMA1 memory address register
			DSA0	DMA0SFR address register
			DSA1	DMA1SFR address register
			DRA0	DMA0 memory address reload register
			DRA1	DMA1 memory address reload register

----- When using 3 or more DMAC channels (DMA2, DMA3) -----

Register bank 1

b15	b0	DCT2(R0)	DMA2 transfer count register
		DCT3(R1)	DMA3 transfer count register
		DRC2(R2)	DMA2 transfer count reload register
		DRC3(R3)	DMA3 transfer count reload register
b23		DMA2(A0)	DMA2 memory address register
		DMA3(A1)	DMA3 memory address register
		DSA2(SB)	DMA2SFR address register
		DSA3(FB)	DMA3SFR address register

When using 3 or more DMAC channels, use Register Bank 1 and Fast Interrupt Register as the registers for DMA2 and DMA3, respectively. Also, when setting values in each register, make sure values are set in the registers enclosed in (). (Note1)(Note2)

High -speed interrupt register

b23	b15	b0	SVF	Flag save register
			DRA2(SVP)	DMA2 memory address reload register
			DRA3(VCT)	DMA3 memory address reload register

Figure 4.2.6 DMAC-related registers

Note1: Before setting DMA2 and DMA3-related registers, always be sure to set Flag Register (FLG)'s register bank specification flag (B) to 1.

Note2: When using DMA2 and DMA3, note that fast interrupts cannot be used. Nor can the registers be saved and restored by register bank switchover in an interrupt handling routine.

Settings when using DMA controller channels 1 to 2 (DMA0, 1)

When using DMAC channels 1 to 2, the following shows an example of how to set the related registers (CPU internal registers and SFR area).

```

;-----Setting DMA0-----
;
MOV.B      #00000011B,DM0SL ; Set cause of DMA0 request
LDC        #32,DRC0         ; Set transfer count
                                ; in DMA0 Transfer Count Reload Register
LDC        #32,DCT0         ; Set transfer count in DMA0 Transfer Count Register
LDC        #0FF0000H,DRA0    ; Set source address of transfer (memory)
                                ; in DMA0 Memory Address Reload Register
LDC        #0FF0000H,DMA0    ; Set destination address of transfer (memory)
                                ; in DMA0 Memory Address Register
LDC        #P6,DSA0         ; Set destination address of transfer (SFR)
                                ; in DMA0 SFR Address Register
LDC        #00001111B,DMD0   ; Set DMA Mode Register 0 and enable DMA transfer
                                ; Unit of transfer: 16 bits
                                ; Direction of transfer: Forward (memory) -> fixed (SFR)
                                ; Transfer mode: Repeat transfer (DMA0 enabled)

```

Figure 4.2.7 Example 1 for setting the DMA controller

Settings when using 3 or more DMA controller channels (DMA2, 3)

When using 3 or more DMAC channels, the following shows an example of how to set the DMA2-related registers (CPU internal registers and SFR area).

```

;-----Setting DMAC channel 3 and those that follow (DMA2 or DMA3)-----
;
;
; FSET      B          ; Set register bank to 1
;
; MOV.B     #00001111B,DM2SL ; Set cause of DMA2 request
; MOV.W     #16,R2          ; Set transfer count
;                               ; in DMA2 Transfer Count Reload Register (R2)
; MOV.W     #16,R0          ; Set transfer count
;                               ; in DMA2 Transfer Count Register (R0)
; LDC       #U0RB,SB        ; Set source address of transfer (SFR)
;                               ; in DMA2 SFR Address Register (SB)
; LDC       #0500H,SVP      ; Set destination address of transfer (memory)
;                               ; in DMA2 Memory Address Reload Register (SVP)
; MOV.L     #0500H,A0        ; Set destination address of transfer (memory)
;                               ; in DMA2 Memory Address Register (A0)
; FCLR      B              ; Return register bank to 0
; LDC       #00001111B,DMD1 ; Set DMA Mode Register 1 and enable DMA transfer
;                               ; Unit of transfer: 16 bits
;                               ; Direction of transfer: Fixed (SFR) -> forward direction (memory)
;                               ; Transfer mode: Repeat transfer (DMA2 enabled)

```

Figure 4.2.8 Example 1 for setting the DMA controller

Note: When using two or less DMAC channels, try using DMA0 and DMA1 as much as possible. If DMA2 and DMA3 are used, Register Bank 1 and fast interrupts become unusable.

4.2.6 Sample Program List 2 (Initial Setting 2)

```

***** Include *****
;
;
;      .INCLUDE      M3800.INC
;
;***** Defined symbol *****
;
;
RAM_TOP      .EQU    000400H      ;Start address of RAM
RAM_END      .EQU    002BFFH      ;End address of RAM
ROM_TOP      .EQU    0FB0000H     ;Start address of ROM
FIXED_VECT_TOP .EQU    0FFFFDCH   ;Start address of fixed vector
SB_BASE      .EQU    00400H       ;Base address of SB relative
FB_BASE      .EQU    00580H       ;Base address of FB relative
ISTACK_SIZE  .EQU    300H         ;Size of interrupt stack area
;
;***** Allocated work RAM area *****
;
;      .SECTION      WORK,DATA
;      .ORG          RAM_TOP
WORKRAM_TOP:
WORK_1:      .BLKB      1
WORK_2:      .BLKB      1
WORKRAM_END:
;
;***** Program area *****
;===== Start up =====
;
;      .SECTION      PROGRAM,CODE      ;Declares section name and section type.
;      .ORG          ROM_TOP           ;Declares start address.
START:
    LDC      #RAM_END + 1,ISP          ;Sets initial value in stack pointer(ISP).
;
;
;      MOV.B   #03H,PRCR               ;Removes protction.
;      MOV.W   #0183H,PM0              ;Sets processor mode register 0 and 1.
;      MOV.W   #2008H,CM0              ;Sets system clock control registers 0 and 1.
;      MOV.B   #12H,MCD                ;Sets main clock divide register.
;      MOV.B   #0,PRCR                 ;Protects all register.
;
;
;      MOV.W   #0,PS0                  ;Sets function select register A0 and A1.
;      MOV.B   #0,PS2                  ;Sets function select register A2.
;      MOV.W   #0,PSL0                 ;Sets function select register B0 and B1.
;      MOV.B   #0,PSL2                 ;Sets function select register B2.
;      MOV.B   #0,PSC                  ;Sets function select register C.
;      BSET    2,PRCR                  ;Removes protection.
;                                          ;(Writes to pin function select register A3 enable)
;      MOV.B   #0,PS3                  ;Sets function select register A3.

```

To prevent the program from going wild when the NMI interrupt (nonmaskable) is generated inadvertently after a reset, set the interrupt stack pointer (ISP) at the beginning of startup procedure.

For protected registers such as Processor Mode Register, remove protection before you set them. Then, when you finished setting the registers, reprotect them.

Because Pin Output Function Select Register 3 is a protected register, remove its protection before setting it. Also, the Pin Output Function Select Register 3 write enable bit (Protect Register bit 3) is reset to 0 by the next write instruction executed after being write-enabled. Therefore, when setting initial values, make sure the value is set in Pin Output Function Select Register 3 immediately after the instruction by which its write enable bit is set to 1.

Addresses 03C9H and 03CBH to 03D3H in the SFR area are reserved for use by products to be developed in the future. Always be sure to initialize addresses 03CBH, 03CEH, 03CFH, 03D2H, and 03D3H with the value "FFH."

```
MOV.B    #03H,DS           ;Sets external data bus width control register.
MOV.B    #85,WCR           ;Sets wait control register.

LDC      #80H,FLG          ;Sets initial value flag register(U=1).
LDC      #(RAM_END - ISTACK_SIZE) + 1,SP ;Sets initial value in stack pointer(USP).

MOV.W    #003FFH,PUR2      ;Connects internal pull-up registers to ports P6 through P10
;
;
MOV.B    #0FFH,03CBH       ;Sets initial value in SFR reserved area.
MOV.W    #0FFFFH,03CEH
MOV.W    #0FFFFH,03D2H
```

```
.SB      SB_BASE           ;Must always be consistent. register value to the assembler.
FB      FB_BASE           ;Declares FB register value to the assembler.

LDC      #SB_BASE,SB       ;Sets initial value in SB register.
LDC      #FB_BASE,FB       ;Sets initial value in FB register.
```

Before RAM access (using SB or FB relative addressing), set the SB and FB registers.

```
MOV.W    #0,R0             ;Clears work RAM area to 0.
MOV.W    #(RAM_END + 1 - RAM_TOP) / 2,R3
MOV.W    #WORKRAM_TOP,A1
SSTR.W
```

===== Main program =====

```
MAIN:
    JSR    INIT             ;Sets initial value in work RAM.
MAIN_10:
    BTST   TA0R             ;Determines TA0 interrupt request flag.
    JNC    MAIN_10
    BCLR   TA0R
    JSR    SUB_TA0          ;Processes timer A0.
;
;
    JMP    MAIN_10
;
```

===== INIT routine =====

----- Initial setting work RAM and ports -----

```
MOV.B    #0FFH,WORK_1
MOV.B    #0FFH,WORK_2
MOV.B    #0FFH,P6
;
```

----- Setting timer A0 -----

```
MOV.B    #01000000B,TA0MR ;Sets timer A0 mode register.
MOV.W    #2500 - 1,TA0    ;Sets timer A0 count value.
BCLR     TA0R             ;Clears timer A0 interrupt request.
BSET     TA0S             ;Timer A0 start counting.
```

After initial setting timer-related registers, set the count start flag.

```

;----- Setting DMA0 -----
MOV.B    #0000011B,DM0SL    ;Sets cause of DMA request.
LDC      #32,DRC0           ;Sets transfer count.
;
LDC      #32,DCT0           ;Sets transfer count in DMA0 transfer count register.
LDC      #0FF0000H,DRA0     ;Sets source address of transfer(memory)
                                ;in DMA0 memory address reload register.
LDC      #0FF0000H,DMA0     ;Sets destination address of transfer(memory)
                                ;in DMA0 memory address register.
LDC      #P6,DSA0           ;Sets destination address of transfer(SFR)
                                ;in DMA0 SFR address register.
LDC      #00001111B,DMD0    ;Sets DMA mode register 0 and enable DMA transfer
                                ;Unit of transfer : 16 bits
                                ;Direction of transfer : Forward(memory)->fixed(SFR)
                                ;Transfer mode : Repeat transfer(DMA0 enable)
INIT_END:
RTS
;===== SUB_TA0 routine =====
SUB_TA0:
MOV.B    WORK_1,R0L
INC.B    R0L
;
;
SUB_TA0_END:
RTS
;===== Dummy interrupt program =====
dummy:
REIT
;
;*****Setting fixed vectors *****
;
.SECTION  F_VECT,ROMDATA
.ORG     FIXED_VECT_TOP
;
.LWORD   dummy    ;Undefined instruction interrupt vector
.LWORD   dummy    ;Overflow(INT0 instruction)interrupt vector
.LWORD   dummy    ;BRK instruction interrupt vector
.LWORD   dummy    ;Address match interrupt vector
.LWORD   dummy    ;Unused
.LWORD   dummy    ;Watchdog timer interrupt vector
.LWORD   dummy    ;Unused
.LWORD   dummy    ;NMI interrupt vector
.LWORD   START    ;Sets reset vector.
.END

```

After setting all of DMA-related registers, enable DMA (set channel transfer mode select bit)

Figure 4.2.9 Description example 2 for initial setting

4.3 Setting when using Interrupts

This section describes the processing and the method of description necessary to execute interrupt handling routines, as well as how to execute multiple interrupts.

Before an interrupt can be generated in the M16C/80 series, all of the following three conditions must be met:

- (1) Interrupt enable flag (I) = 1 (interrupt enabled)
- (2) $IPL < \text{Software interrupt priority level of the interrupt generated}$
- (3) Interrupt request bit for the interrupt used = 1 (interrupt requested)

In addition to the above three conditions, following processing are required before an interrupt handling routine can be executed:

- (1) Set Interrupt Table Register (INTB)
- (2) Set variable/fixed vectors
- (3) Set Interrupt Control Register
- (4) Enable interrupt enable flag (I)
- (5) Save and restore registers in interrupt handling routine

4.3.1 Setting Interrupt Table Register(INTB)

Since the vector tables for interrupts from internal peripheral functions in the M16C/80 series are variable, it is necessary to set the start address of the vector using Interrupt Table Register (INTB) before using interrupts.

The 256 bytes of space from the address specified by the Interrupt Table Register is the variable vector area, with each vector consisting of 4 bytes. Each vector is assigned a software interrupt number, together comprising 64 vectors from 0 to 63.

For setup examples, refer to Section 4.3.6, "Sample List 3 (Using Interrupts)."

4.3.2 Setting Variable/Fixed Vectors

When an interrupt occurs, the program jumps to the address that has been set for each cause of interrupt. The part of memory in which this jump address is set is referred to as the "interrupt vector." To set interrupt vectors, register the start address of each interrupt handler program in the variable/fixed vector table. For an example of how the vectors actually are registered, refer to Section 4.3.6, "Sample Program List 3 (Software Interrupt)".

Variable Vector Table

The variable vector table is a 256-byte interrupt vector table that starts from the address specified by Interrupt Table Register (INTB). The vector table can be located anywhere in memory space except the SFR area. One vector consists of 4 bytes, with each vector assigned software interrupt numbers 0 to 63.

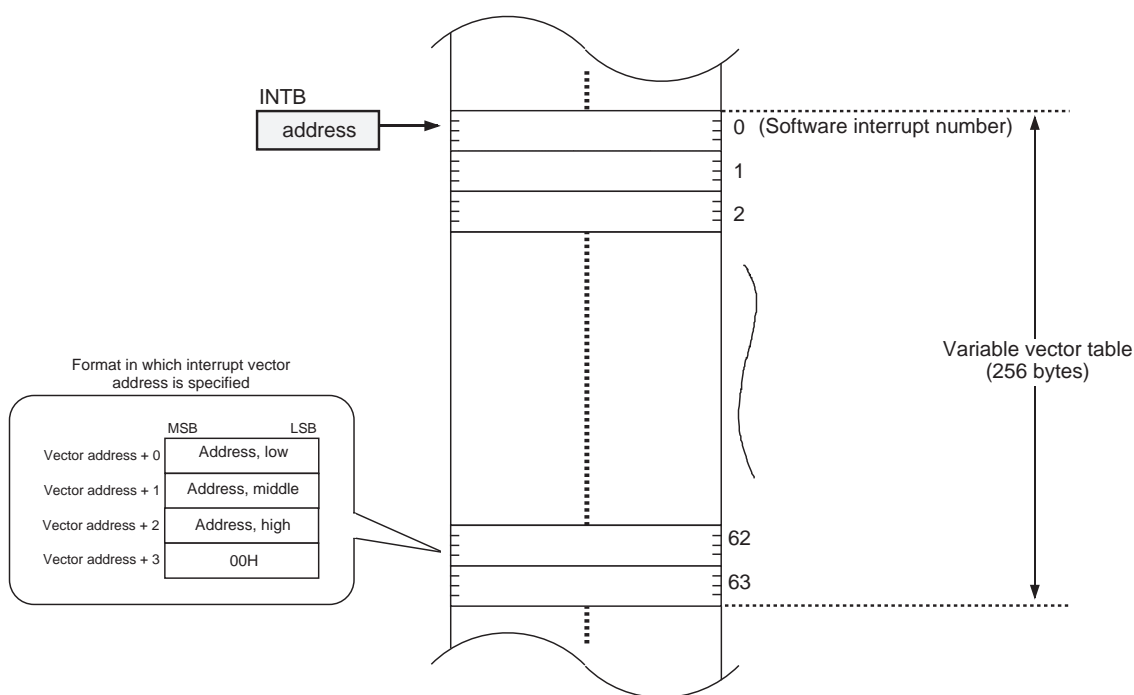


Figure 4.3.1 Variable vector table

4.3.3 Setting Interrupt Control Register

Bits 0-2 of each interrupt control register can be used to set the interrupt priority level of each interrupt. When level = 0, the effect is the same as interrupts being disabled, so make sure the priority levels set are equal to or greater than 1. The Interrupt Control Register bit 3 serves as an interrupt request flag. This flag is 0 after a reset, but because this flag for some external pin interrupt may have been set to 1, be sure to clear it to 0 before enabling the interrupt enable flag (I flag). For setup examples, refer to Section 4.3.6, "Sample List 3 (Using Interrupts)."

For details about the bit configuration and priority levels of each interrupt control register, consult the user's manual supplied with your microcomputer.

4.3.4 Enabling Interrupt Enable Flag(I flag)

Immediately after power-on and after a reset, interrupts are in disabled state. Therefore, interrupts must be enabled in the program. This can be accomplished by setting the Flag Register (FLG)'s I flag to 1. Because interrupts are enabled at the same time the I flag is set to 1, caution must be used to prevent the program from going wild. To this end, always be sure to enable the I flag after making initial settings, and not at the beginning of the program.

4.3.5 Saving and Restoring Registers in Interrupt Handler Routine

When an interrupt is accepted, the following resources are automatically saved to the stack. For details on how they are saved and restored to and from the stack, refer to Section 4.5.2, "Stack Area."

- (1) Contents of PC (program counter)
- (2) Contents of FLG (flag register)

Always be sure to use the REIT instruction to return from the interrupt handler routine. After the interrupt processing is completed, this instruction restores the registers, return address, etc. from the stack, thus Except for automatically saved registers, if there are any registers that are likely to be modified in the interrupt handling routine (e.g., registers used in interrupt handling), save them to the stack in software. For an example of how to save and restore registers in an interrupt handling routine, refer to Figures 4.3.2 and 4.3.3.

Methods for Saving and Restoring Registers

If in addition to the automatically saved registers there is any register which is used in the interrupt handler routine and, therefore, whose previous content needs to be retained, save it to the stack area in software. There are two methods for saving and restoring this register. The following shows the processing procedure for each method.

There are following two methods for saving/restoring registers.

(1) Saving and restoring by push/pop instructions

(1a) Saving registers individually

```
PUSH.B   R0L
PUSH.W   R1
```

(1b) Restoring registers individually

```
POP.B    R0L
POP.W    R1
```

(2a) Saving registers collectively

```
PUSHM    R0,R1,R2,R3,A0,A1
```

(2b) Restoring registers collectively

```
POPM     R0,R1,R2,R3,A0,A1
```

(2) Saving and restoring by register bank switchover

This method is effective when a reduction in interrupt handling overhead time is desired.

(a) Using register bank 1

```
FSET     B
```

(b) Using register bank 0

```
FCLR     B
```

Description of Interrupt Handling Program(1)

Figure 4.3.2 shows an example for writing an interrupt handling program.

```

;*****Saving and restoring registers individually*****
INT_A0:
    PUSH.B    R0L    ; Saves R0L.
    PUSH.B    R1L    ; Saves R1L.
    PUSH.W    R2      ; Saves R2.
    .
    .
    Interrupt handling
    .
    .
    POP.W R2      ; Restores R2.
    POP.B R1L    ; Restores R1L.
    POP.B R0L    ; Restores R0L.
    ;
    REIT          ; Returns from interrupt.

;***** Saving and restoring registers collectively*****
INT_A1:
collectively.    PUSHM    R0,R1,R2,R3    ; Saves registers R0, R1, R2, and R3
    .
    .
    Interrupt handling
    .
    .
collectively.    POPM    R0,R1,R2,R3      ; Restores registers R0, R1, R2, and R3
    REIT          ; Returns from interrupt.
    ;

```

If registers are saved individually, be sure when restoring them to reverse the order in which they were saved.

Figure 4.3.2 Saving and restoring registers in interrupt handling

Note: If both register banks 0 and 1 are used in the main program, the method for saving and restoring registers by register bank switchover cannot be used.

Description of Interrupt Handling Program(2)

If high-speed interrupt acknowledgment is desired, use the register bank switchover shown below^(Note).

Registers (R0, R1, R2, R3, A0, A1, SB, and FB) can be saved/restored by one instruction, "FSET B" or "FCLR B" (number of execution cycles: 1).

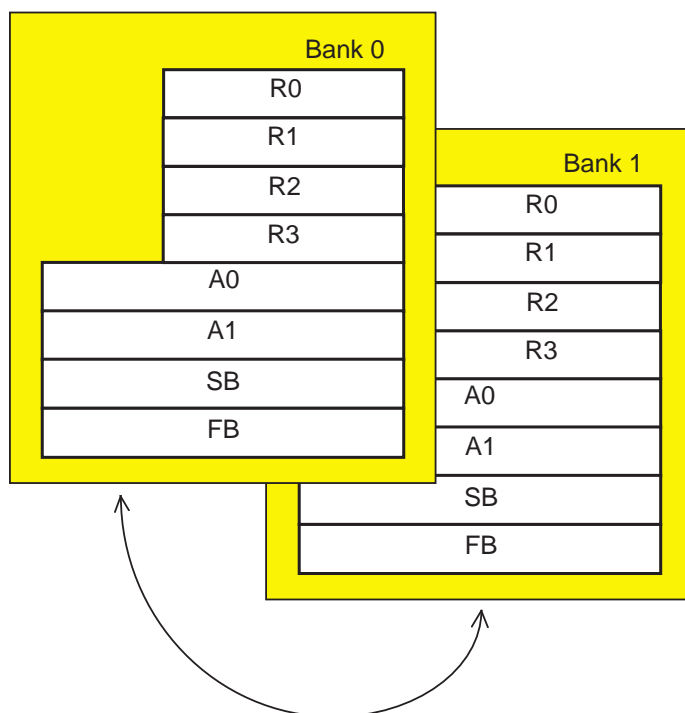
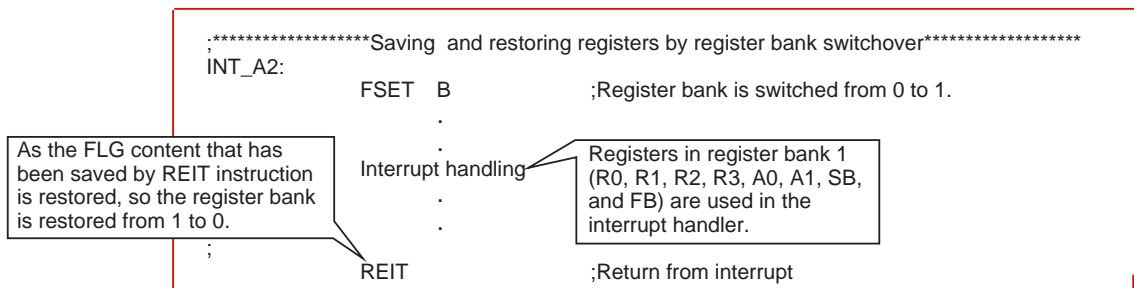


Figure 4.3.3 Saving and restoring registers by register bank switchover

Note: If register banks 0 and 1 both are being used in the main program, registers cannot be saved/restored by register bank switchover.

4.3.6 Sample Program List 3 (Using interrupts)

```

***** Include*****
;
;
;      .INCLUDE          M30800.INC
;
;
***** Define symbol *****
;
;
RAM_TOP          .EQU    000400H      ; Start address of RAM
RAM_END          .EQU    002BFFH      ; End address of RAM
ROM_TOP          .EQU    0FE0000H     ; Start address of ROM
FIXED_VECT_TOP   .EQU    0FFFFDCH     ; Start address of fixed vector
SB_BASE          .EQU    00400H       ; Base address for SB relative
FB_BASE          .EQU    00580H       ; Base address for FB relative
ISTACK_SIZE      .EQU    300H         ; Size of interrupt stack area
;
***** Allocated work RAM area *****
;
;
      .SECTION          WORK,DATA
      .ORG              RAM_TOP
WORKRAM_TOP:
WORK_1:          .BLKW      1
ANS_L:           .BLKW      1
ANS_H:           .BLKW      1
WORKRAM_END:
;
;
***** Program area *****
;
===== Startup=====
;
;
      .SECTION          PROGRAM,CODE      ; Declares section name and section type.
      .ORG              ROM_TOP          ; Declares start address.
START:
      LDC               #RAM_END+1,ISP    ; Sets initial value in stack pointer (ISP).
;
;
      MOV.B             #03H,PRCR         ; Removes protection.
      MOV.W             #0183H,PM0        ; Sets processor mode register 0 and 1.
      MOV.W             #2008H,CM0        ; Sets system clock control registers 0 and 1.
      MOV.B             #12H,MCD         ; Sets main clock divide register.
      MOV.B             #0,PRCR          ; protects all registers.
;
;
      MOV.W             #0,PS0            ; Sets function select register A0 and A1.
      MOV.B             #0,PS2            ; Sets function select register A2.
      MOV.W             #0,PSL0          ; Sets function select register B0 and B1.
      MOV.B             #0,PSL2          ; Sets function select register B2.
      MOV.B             #0,PSC           ; Sets function select register C.
      BSET              2,PRCR            ; Removes protection.
;
      MOV.B             #0,PS3            ; (Write to pin function select register A3 enabled.)
;
;
      MOV.B             #0,PS3            ; Sets function select register A3.
;
;

```

```

MOV.B      #03H,DS      ; Sets External data bus width control register.
MOV.B      #85H,WCR      ; Sets wait control register.
;
LDC  #80H,FLG      ; Sets initial value in flag register(U = 1).
LDC  #(RAM_END-ISTACK_SIZE)+1,SP      ; Sets initial value in stack pointer(USP).
LDC  #VECT_TOP,INTB      ; Sets initial interrupt table register.
;
MOV.W      #003FFH,PUR2      ; Connects internal pull-up registers to ports P6 through P10
;
MOV.B      #0FFH,03CBH      ; Sets initial value in SFR reserved area.
MOV.W      #0FFFFH,03CEH
MOV.W      #0FFFFH,03D2H
;
.SB  SB_BASE      ; Declares SB register value to the assembler.
.FB  FB_BASE      ; Declares FB register value to the assembler.
LDC  #SB_BASE,SB      ; Sets initial value in SB register.
LDC  #FB_BASE,FB      ; Sets initial value in FB register.
;
MOV.W      #0,R0      ; Clears work RAM area to 0.
MOV.W      #(RAM_END+1 - RAM_TOP)/2,R3
MOV.W      #WORKRAM_TOP,A1
SSTR.W
;=====Main program =====
MAIN:
    JSR      INIT      ; Sets initial value in work RAM.
    FSET     I      ; Enable interrupts.
MAIN_10:
    MOV.W    #5,WORK_1
    MULU.W   WORK_1,ANS_L
    ;
    ;
    ;
    JMP      MAIN_10
;
;===== INIT routine =====
INIT:
    MOV.W    #0,WORK_1
    MOV.W    #0,ANS_L
    MOV.W    #0,ANS_H
    ;
    MOV.B    #01000000B,TA0MR      ; Sets timer A0 mode register.
    MOV.W    #2500-1,TA0      ; Sets timer A0 count value.
    MOV.B    #00000111B,TA0IC      ; Sets interrupt priority level for timer A0
    ; (level : 7) and clears interrupt request bit.
    BCLR     TA0R      ; Clears timer A0 interrupt request bit.
    BSET     TA0S      ; Timer A0 starts counting.
INIT_END:
    RTS

```

Interrupts are enabled after making initial settings.

For interrupts to be generated, the priority level must be set to any value equal to or greater than 1.

```

;
;===== TA0 interrupt handling routine =====
INT_TA0:
        PUSHM      R0,R1,R2,R3,A0,A1    ;Saves registers.
;
;        .
;
;        .
;        Program
;
;        .
;
;        .
;
        POPM      R0,R1,R2,R3,A0,A1    ;Restores registers.
INT_TA0_END:
;===== Dummy interrupt program =====
dummy:
        REIT
;***** Setting variable vector table ****
;
;        .SECTION    VECT,ROMDATA
;        .ORG        VECT_TOP
;
;        .LWORD      dummy              ; BRK instruction interrupt vector
;
;        .ORG        VECT_TOP + (8 * 4 )
;        .LWORD      dummy              ; DMA0 interrupt vector.
;        .LWORD      dummy              ; DMA1 interrupt vector.
;        .LWORD      dummy              ; DMA2 interrupt vector.
;        .LWORD      dummy              ; DMA3 interrupt vector.
;        .LWORD      INT_TA0            ; Sets start address of interrupt handler for
;                                         ; Timer A0 interrupt vector.
;        .LWORD      dummy              ; Timer A1 interrupt vector.
;        .LWORD      dummy              ; Timer A2 interrupt vector.
;        .LWORD      dummy              ; Timer A3 interrupt vector.
;        .LWORD      dummy              ; Timer A4 interrupt vector.
;        .LWORD      dummy              ; UART 0 transfer interrupt vector.
;        .LWORD      dummy              ; UART 0 receive interrupt vector.
;        .LWORD      dummy              ; UART 1 transfer interrupt vector.
;        .LWORD      dummy              ; UART 1 receive interrupt vector.
;        .LWORD      dummy              ; Timer B0 interrupt vector.
;        .LWORD      dummy              ; Timer B1 interrupt vector.
;        .LWORD      dummy              ; Timer B2 interrupt vector.
;        .LWORD      dummy              ; Timer B3 interrupt vector.
;        .LWORD      dummy              ; Timer B4 interrupt vector.
;        .LWORD      dummy              ; INT5 interrupt vector.
;        .LWORD      dummy              ; INT4 interrupt vector.
;        .LWORD      dummy              ; INT3 interrupt vector.
;        .LWORD      dummy              ; INT2 interrupt vector.
;        .LWORD      dummy              ; INT1 interrupt vector.
;        .LWORD      dummy              ; INT0 interrupt vector.

```

To return from interrupt, use REIT instruction, and not RTS instruction.

Because vector numbers 1 to 7 are assigned internal peripheral function interrupts, this statement sets an address 32 bytes (4 bytes x 8) forward from the start address of the variable interrupt vector, that is, the address of vector number 8.

For unused interrupts, set their jump addresses in dummy processing (REIT instruction only) to prevent the program from going wild when an unused interrupt occurs.

```

.LWORD    dummy    ; Timer B5 interrupt vector.
.LWORD    dummy    ; UART2 transfer / NACK interrupt vector.
.LWORD    dummy    ; UART2 receive / ACK interrupt vector.
.LWORD    dummy    ; UART3 transfer / NACK interrupt vector.
.LWORD    dummy    ; UART3 receive / ACK interrupt vector.
.LWORD    dummy    ; UART4 transfer / NACK interrupt vector.
.LWORD    dummy    ; UART4 receive / ACK interrupt vector.
.LWORD    dummy    ; Bus collision detection / start,stop
.LWORD    dummy    ; Condition(UART 2) interrupt vector.
.LWORD    dummy    ; Bus collision detection / start,stop
.LWORD    dummy    ; Condition(UART 3) interrupt vector.
.LWORD    dummy    ; Bus collision detection / start,stop
.LWORD    dummy    ; Condition(UART 4) interrupt vector.
.LWORD    dummy    ; A-D interrupt vector.
.LWORD    dummy    ; Key- input interrupt vector.

;
; ***** Setting fixed vectors *****
;
;
.SECTION   F_VECT,ROMDATA
.ORG      FIXED_VECT_TOP

.LWORD    dummy    ;Undefined instruction interrupt vector
.LWORD    dummy    ;Overflow (INTO instruction) interrupt vector.
.LWORD    dummy    ;BRK instruction interrupt vector.
.LWORD    dummy    ;Address match interrupt vector
.LWORD    dummy    ;Unused
.LWORD    dummy    ;Watchdog timer interrupt vector
.LWORD    dummy    ;Unused
.LWORD    dummy    ;NMI interrupt vector
.LWORD    dummy    ;Sets Reset vector

;
;
.END

```

Figure 4.3.4 Sample program 3(Using interrupt)

4.3.7 ISP and USP

The M16C/80 series has two stack pointers: an interrupt stack pointer (ISP) and a user stack pointer (USP). Use of these stack pointers is selected by the U flag.

(1) ISP is used when U = 0

Registers are saved and restored to and from the address indicated by ISP.

(2) USP is used when U = 1

Registers are saved and restored to and from the address indicated by USP.

Use the ISP when programming in assembly language (not using high-level languages or OS). Although the USP may be used, caution is required when using peripheral I/O interrupts. For details, refer to "Relationship between software interrupt numbers and stack pointers" in the next page.

Regarding assignments of software interrupt numbers

The M16C/80 series has software interrupt numbers from 0 to 63. Numbers 8 through 43 are reserved for peripheral I/O interrupts(note 1). Therefore, remaining numbers 0 through 7 and 44 through 63 can be assigned software interrupts (INT instructions(note 2)). However, for the purpose of application, software interrupt numbers 32 through 63 are assigned to software interrupts used by the OS, etc. (numbers 48 through 63 used by M16C/80 real-time monitor (MR308), for example). When using an OS in your system, use software interrupt numbers 0 through 7 only.

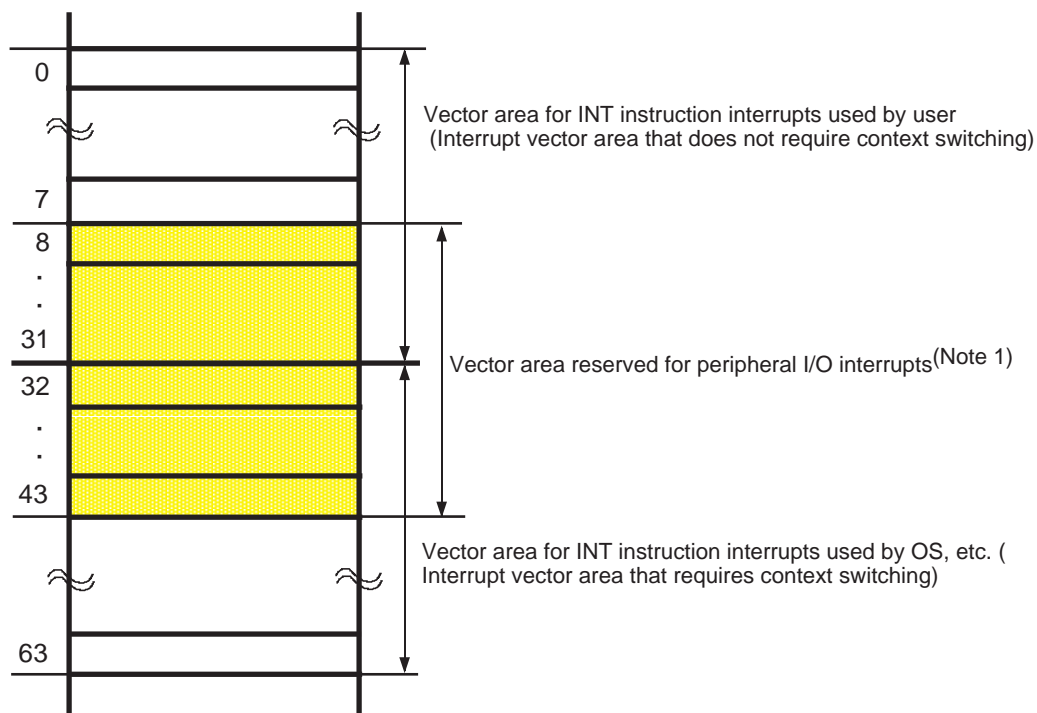


Figure 4.3.5 Interrupt number assignments

Note 1: This varies with the type of microcomputer used. Please consult the user's manual supplied with your microcomputer.

Note 2: The program branches to the address that is stored in the interrupt number specified by the INT instruction operand.

Relationship between software interrupt numbers and stack pointers

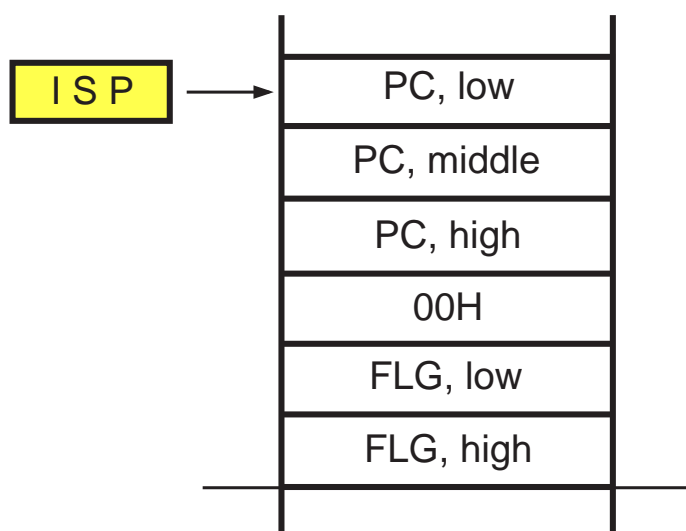
(1) When a peripheral I/O interrupt or an INT instruction interrupt using software interrupt numbers 0 through 31 occurs

- (a) The CPU reads address 000000H to get interrupt information (interrupt number, interrupt request level) and then clears the interrupt request bit for the accepted interrupt to 0.
- (b) The FLG register content is saved to the CPU's internal temporary register.
- (c) Flags U, I, and D of the FLG register are cleared.

Thus, by operation in (c)...

- (i) The stack pointer is forcibly made the interrupt stack pointer (ISP).
- (ii) Multiple interrupts are disabled.
- (iii) Debug mode is cleared (not single-stepped).
- (d) The CPU's internal temporary register (to which FLG has been saved) and PC register contents are saved to the stack area.
- (e) The interrupt priority level of the accepted interrupt is set in the processor Interrupt Priority Level (IPL).
- (f) The address written in the interrupt vector is transferred to the PC register.

<Stack status after accepting interrupt request>



<FLG status after accepting interrupt request>

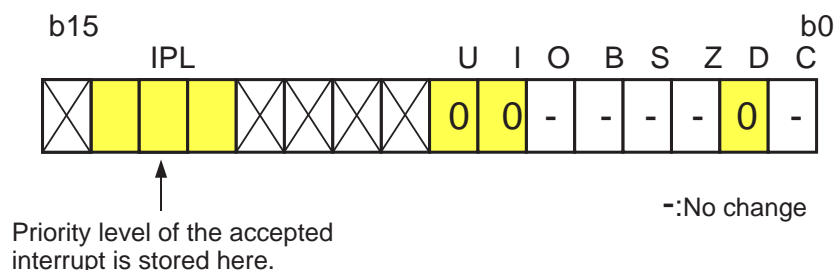
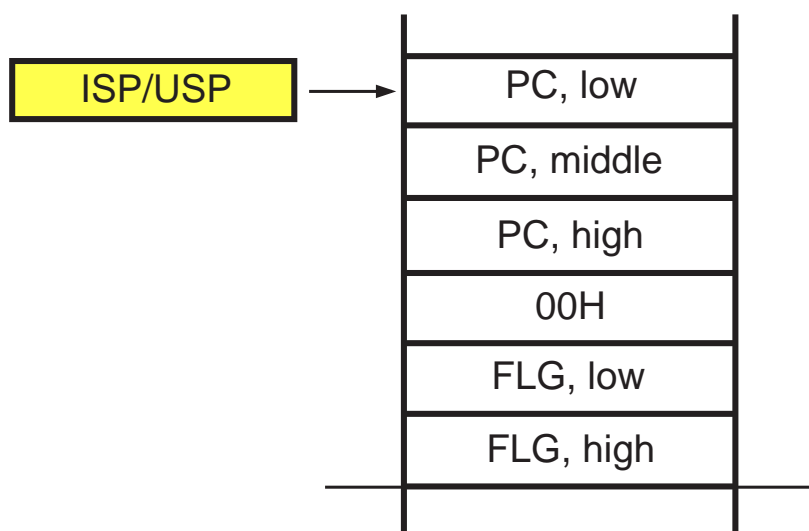


Figure 4.3.6 When a peripheral I/O interrupt or an INT instruction interrupt using software interrupt numbers 0 through 31 occurs

- (a) The CPU reads address 000000H to get interrupt information (interrupt number, interrupt request level) and then clears the interrupt request bit for the accepted interrupt to 0.
- (b) The FLG register content is saved to the CPU's internal temporary register.
- (c) Flags I and D of the FLG register are cleared.

- (i) The stack pointer used here is one that was active when the interrupt occurred.
- (ii) Multiple interrupts are disabled.
- (iii) Debug mode is cleared (not single-stepped).

- <Stack status after accepting interrupt request>



b15				IPL					U	I	O	B	S	Z	D	b0	
								-	0	-	-	-	-	-	0	-	

- ∴ No change

Figure 4.3.7 When an INT instruction interrupt using software interrupt numbers 32 through 63 occurs

4.3.8 Multiple Interrupts

In the M16C/80 series, once an interrupt request is accepted, the interrupt enable flag (I) is automatically cleared to 0 (interrupts disabled), so that no other interrupts are accepted until processing of the accepted interrupt is completed. Therefore, if another interrupt needs to be generated while an interrupt is being serviced, this can be accomplished by setting the interrupt enable flag (I) to 1 (interrupts enabled) in the interrupt handling routine.

Example of Multiple Interrupt Execution

As an execution example for multiple interrupts, or interrupts generated while an interrupt being serviced, an execution flow is shown in Figure 4.3.8 where multiple interrupts (1), (2), and (3) occur.

- (1) Interrupt 1 occurs when executing the main routine
- (2) Interrupt 2 occurs when executing interrupt 1
- (3) Interrupt 3 occurs when executing interrupt 2

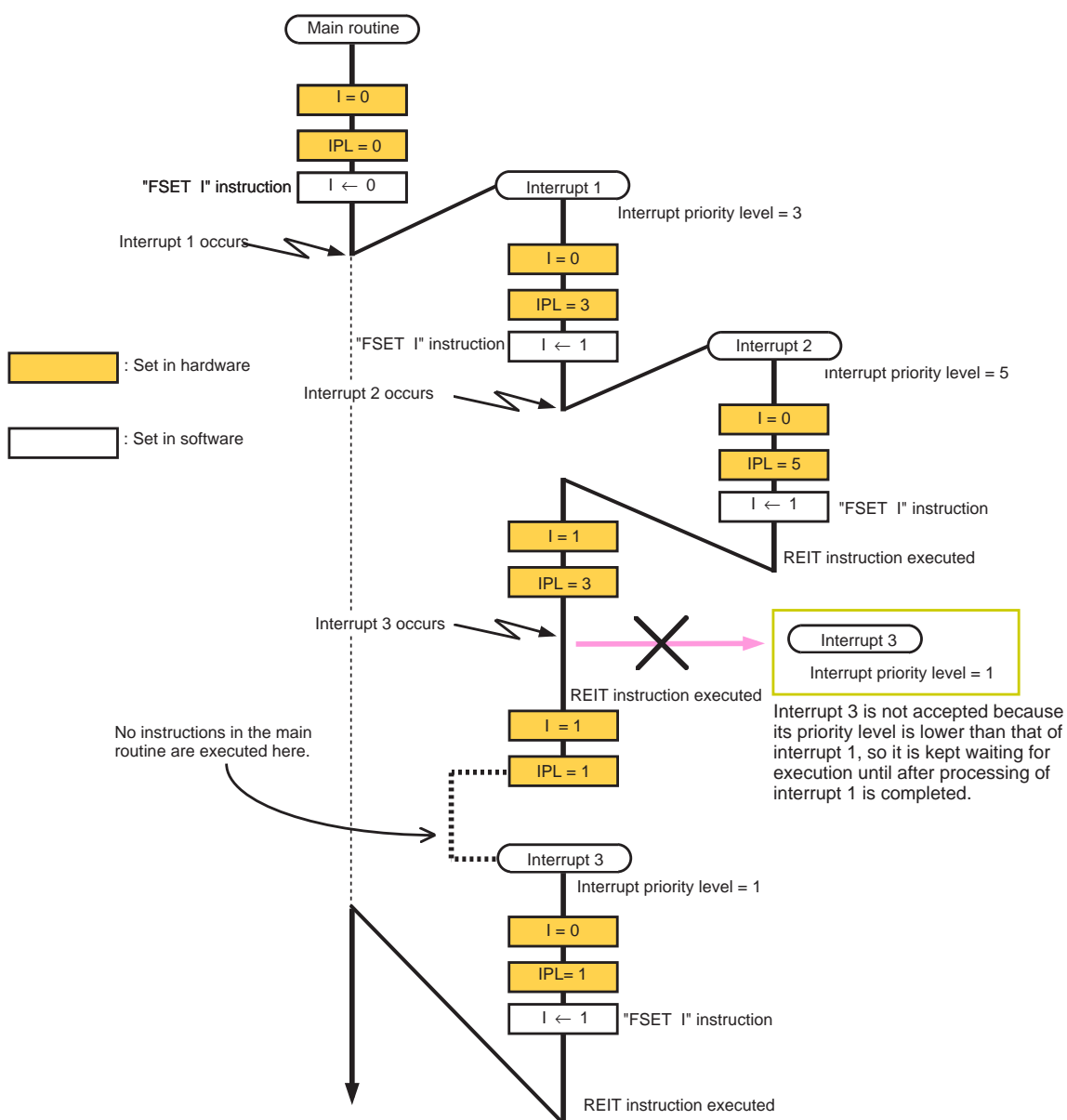


Figure 4.3.8 Execution example of multiple interrupts

4.3.9 High-speed interrupts

High-speed interrupts refer to an interrupt for which the interrupt acknowledgment (interrupt handling sequence) can be executed in 5 cycles and return from which can be executed in 3 cycles.

High-speed interrupts are handled in such a way that when an interrupt is accepted, the flag register (FLG) and program counter (PC) respectively are saved to the CPU's internal registers, Save Flag Register (SVF) and Save PC Register (SVP), and the program is executed from the address indicated by the Vector Register (VCT).

High-speed interrupts become usable by setting the fast interrupt specification bit^(Note 1) to 1, and the interrupt^(Note 2) for which the software interrupt priority level has been set to 7 is handled as a fast interrupt. The diagram below shows how a fast interrupt operates.

High-speed interrupt acknowledgment/return operations

Because high-speed interrupts are one whose interrupt sequence is shortened, use "bank switchover" to save/restore registers, and in the high-speed interrupt handler routine, use "register bank 1" as a dedicated register, as much as possible^(Note 3).

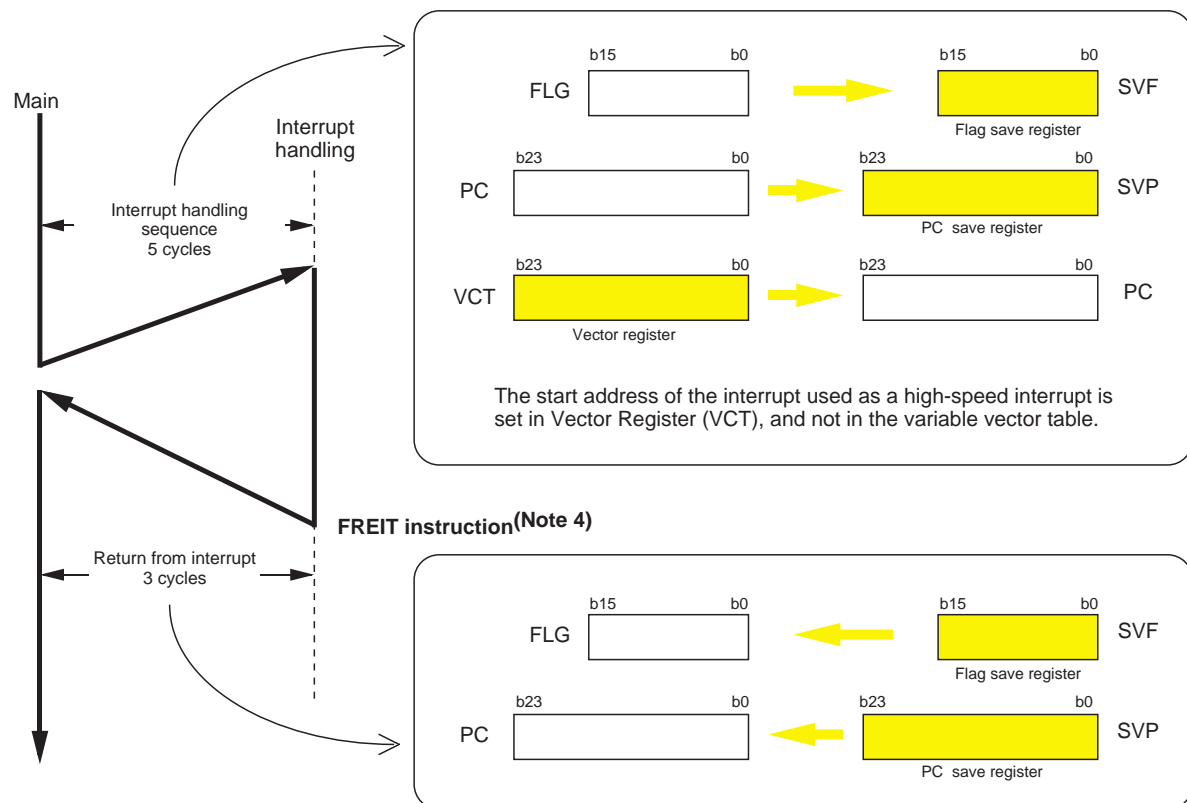


Figure 4.3.9 Operation of a high-speed interrupt

Note 1: This bit is assigned to bit 3 of the Return Priority Register.

Note 2: Because only one interrupt at a time can be set as a fast interrupt, make sure there is only one interrupt whose interrupt priority level = 7.

Note 3: In this case, register bank 1 cannot be used in the main routine.

Note 4: Execute the FREIT instruction to return from the fast interrupt routine.

Program description example when using high-speed interrupts

```

*****Include*****
;
; .INCLUDE      M3800.INC
;*****Program area*****
;
; .SECTION      PROGRAM, CODE      ; Declares section name and section type.
; .ORG          ROM_TOP            ; Declares start address.
START:
;
;      .
;
;      .
;      Initial setting CPU
;
;      .
;
;=====Main program=====
MAIN:
      JSR        INIT              ; Sets initial value for work RAM.
      FSET       I                 ; Enables interrupts.
MAIN_10:
      MOV.W      WORK_1,R0
;
;      .
;
      JMP        MAIN_10
;
;=====INIT routine=====
INIT:
      MOV.W      #0,WORK_1
      MOV.W      #0,WORK_2
      MOV.B      #00000111B,INT0IC ; Sets priority level of INT0 pin interrupt
                                      ; (level: 7), active edge (falling edge), and clear
                                      ; interrupt request bit
      LDC        #INT_INT0,VCT     ; Sets vector register.
      BSET       3,RLVL            ; Sets return priority register.
INIT_END:
      RTS
;=====INT0 interrupt handling routine=====
INT_INT0:
      FSET       B                 ; Saves registers by register bank swichover.
;
;      .
;      Program
;      .
INT_INT0_END:
      FREIT                          ; Returns from high-speed interrupt.
;

```

When using fast interrupts, note that there is only one interrupt whose interrupt priority level can be set to 7.

Set the start address of the interrupt handler routine in the vector register (fast interrupt-only interrupt vector).

Set the fast interrupt specification bit to 1, so the interrupt whose interrupt priority level = 7 is used as a high-speed interrupt.

Use the FREIT instruction to return from a fast interrupt.

By using register bank swichover to save registers, the interrupt acknowledgement time can further be reduced.

```

;===== Dummy interrupt program =====
dummy:
    REIT
;***** Setting variable vector table *****
;
;
; .SECTION      VECT,ROMDATA
; .ORG          VECT_TOP
;
; .LWORD        dummy                ; BRK instruction interrupt vector
;
; .ORG          VECT_TOP + ( 8 * 4 )
; .LWORD        dummy                ; DMA0 interrupt vector.
; .LWORD        dummy                ; DMA1 interrupt vector.
; .LWORD        dummy                ; DMA2 interrupt vector.
; .LWORD        dummy                ; DMA3 interrupt vector.
; .LWORD        dummy                ; Timer A0 interrupt vector.
; .LWORD        dummy                ; Timer A1 interrupt vector.
; .LWORD        dummy                ; Timer A2 interrupt vector.
; .LWORD        dummy                ; Timer A3 interrupt vector.
; .LWORD        dummy                ; Timer A4 interrupt vector.
; .LWORD        dummy                ; UART 0 transfer interrupt vector.
; .LWORD        dummy                ; UART 0 receive interrupt vector.
; .LWORD        dummy                ; UART 1 transfer interrupt vector.
; .LWORD        dummy                ; UART 1 receive interrupt vector.
; .LWORD        dummy                ; Timer B0 interrupt vector.
; .LWORD        dummy                ; Timer B1 interrupt vector.
; .LWORD        dummy                ; Timer B2 interrupt vector.
; .LWORD        dummy                ; Timer B3 interrupt vector.
; .LWORD        dummy                ; Timer B4 interrupt vector.
; .LWORD        dummy                ; INT5 interrupt vector.
; .LWORD        dummy                ; INT4 interrupt vector.
; .LWORD        dummy                ; INT3 interrupt vector.
; .LWORD        dummy                ; INT2 interrupt vector.
; .LWORD        dummy                ; INT1 interrupt vector.
; .LWORD        dummy                ; INT0 interrupt vector.
; .LWORD        dummy                ; Timer B5 interrupt vector.
; .LWORD        dummy                ; UART2 transfer / NACK interrupt vector.
; .LWORD        dummy                ; UART2 receive / ACK interrupt vector.
; .LWORD        dummy                ; UART3 transfer / NACK interrupt vector.
; .LWORD        dummy                ; UART3 receive / ACK interrupt vector.
; .LWORD        dummy                ; UART4 transfer / NACK interrupt vector.
; .LWORD        dummy                ; UART4 receive / ACK interrupt vector.
; .LWORD        dummy                ; Bus collision detection / start,stop
; .LWORD        dummy                ; Condition(UART 2) interrupt vector.
; .LWORD        dummy                ; Bus collision detection / start,stop
; .LWORD        dummy                ; Condition(UART 3) interrupt vector.
; .LWORD        dummy                ; Bus collision detection / start,stop
; .LWORD        dummy                ; Condition(UART 4) interrupt vector.
; .LWORD        dummy                ; A-D interrupt vector.
; .LWORD        dummy                ; Key- input interrupt vector.

```

For the interrupt which has been set as a high-speed interrupt, do not set the start address of the interrupt handler routine in the variable interrupt vector.

```
;
;***** Setting fixed vectors *****
;
;      .SECTION      F_VECT,ROMDATA
;      .ORG          FIXED_VECT_TOP
;
;      .LWORD        dummy          ;Undefined instruction interrupt vector
;      .LWORD        dummy          ;Overflow (INTO instruction) interrupt vector.
;      .LWORD        dummy          ;BRK instruction interrupt vector.
;      .LWORD        dummy          ;Address match interrupt vector
;      .LWORD        dummy          ;Unused
;      .LWORD        dummy          ;Watchdog timer interrupt vector
;      .LWORD        dummy          ;Unused
;      .LWORD        dummy          ;NMI interrupt vector
;      .LWORD        START          ;Sets Reset vector
;
;      .END
```

Figure 4.3.10 Program example when using a high-speed interrupt

4.4 Dividing Source File

Write the program separately in several source files. This helps to make your program put in order and easily readable. Furthermore, since the program can be assembled separately one file at a time, it is possible to reduce the assemble time when correcting the program. This section explains how to divide the source file.

4.4.1 Concept of Sections

A program written in the assembly language generally consists of a work area, program area, and constant data area. When the source file (***.AS30) is assembled by the assembler (as308), relocatable module files (***.R30) are generated. The relocatable module files contain one or more of these areas. A section is the name that is assigned to each of these areas. Consequently, a section can be considered to be the name that is assigned to each constituent element of the program.

Note that the assembler (as308) requires that even in the case of the absolute file, there must always be at least one section specified in one file.

Functions of Sections

When linking the source files, the areas of the same section name are located at contiguous addresses sequentially in order of specified files. Furthermore, the start address of each section can be specified when linking. This means that each section can be relocated any number of times without having to change the source program. Figure 4.4.1 shows an example of how sections actually are located in memory.

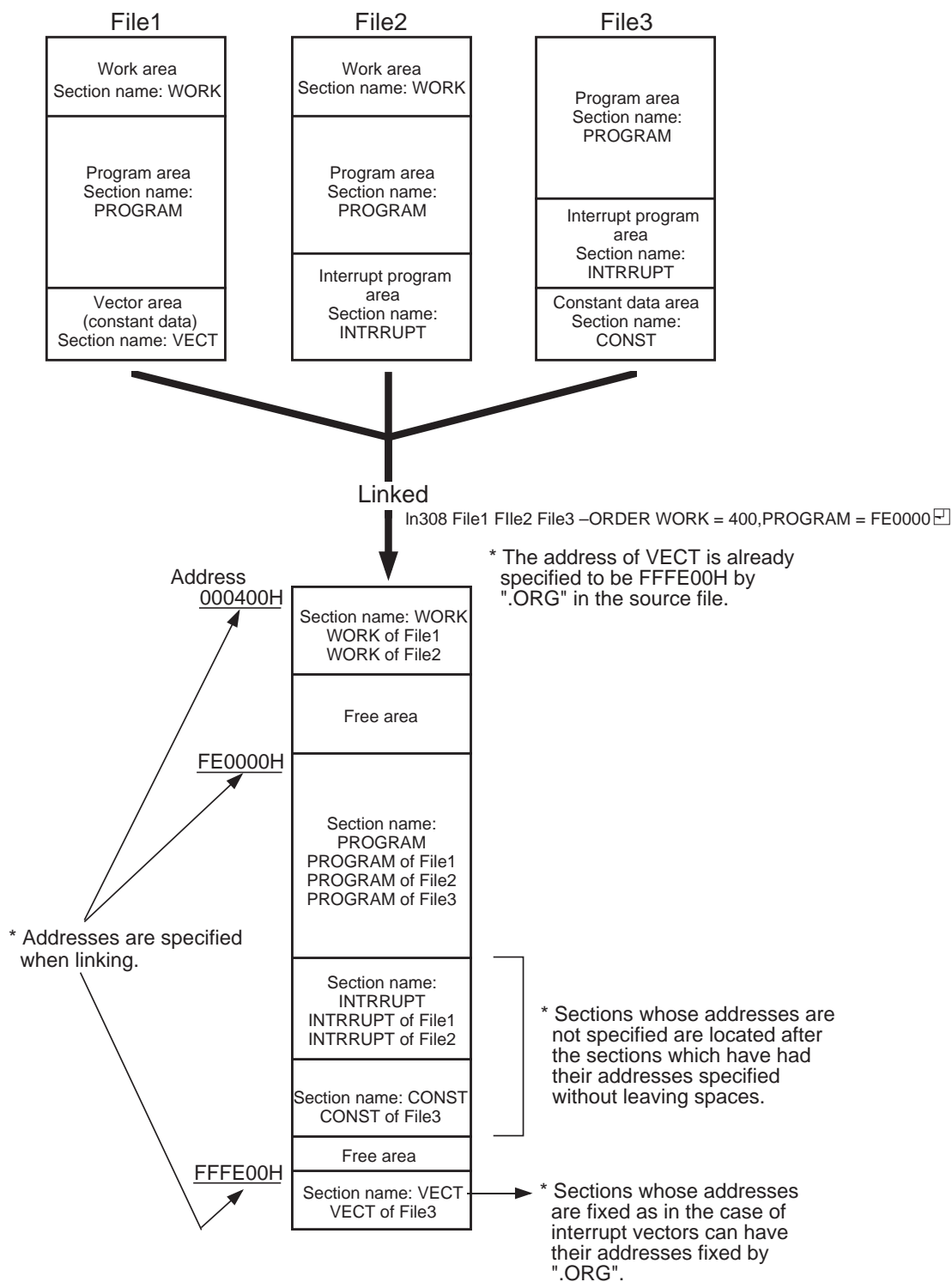


Figure 4.4.1 Example of sections located in memory

4.4.2 Example of program description in divided files

The as308 used in this manual is a relocatable assembler. When using a relocatable assembler, it is normally desirable to write the program source separately in several files. The following lists the advantages that can be obtained by dividing the source file:

(1) Shared program and data

Data exchanges between development projects are facilitated, making it possible to reuse only a necessary part from existing software.

(2) Reduced assemble time

When modifying or correcting the program, only the modified or corrected file needs to be reassembled. This helps to reduce the assemble time.

The following explains how to write the source program in cases when the file is divided into three (definition, main program, and subroutine processing).

Division Example 1: Definition (WORK.A30)

Write definitions of the work RAM area and data table in file 1.

```

*****
;
; File 1 (WORK.A30)
;
*****
===== Allocation of work RAM area=====
;
;
; .SECTION      WORK,DATA
; .ORG          RAM_TOP
; .GLB          WORK_1,WORK_2,WORK_3,WORK_4      ; Processed as global label.
; .GLB          DATA_TABLE                      ; Processed as global label.
; .BTGLBW1_b0,W2_b1                             ; Processed as global bit symbol.
;
GLOBAL_WORK_TOP:
WORK_1:          .BLKB 1
WORK_2:          .BLKB 1
WORK_3:          .BLKB 1
WORK_4:          .BLKB 1
GLOBAL_WORK_END:
W1_b0            .BTEQU 0,WORK_1
W2_b1            .BTEQU 1,WORK_2
;
;
; =====Fixed data area=====
;
; .SECTION      CONSTANT,ROMDATA
; .ORG          CONST_TOP
;
DATA_TABLE:
; .BYTE 12H
; .BYTE 34H
; .BYTE 56H
; .BYTE 78H
DATA_TABLE_END:
;
; .END

```

In order for work RAM and labels to be referenced from another file, declare global labels using .GLB.

In order for bit symbol defined by .BTEQU to be referenced from another file, declare global symbols using .BTGLB.

; Allocates work RAM area.

; Defines bit symbols.

; Sets 1-byte data.

Figure 4.4.2 Divided file 1 (WORK.A30)

Division Example 2: Main Program (MAIN.A30)

Write the main program("PROGRAM" section) in file 2.

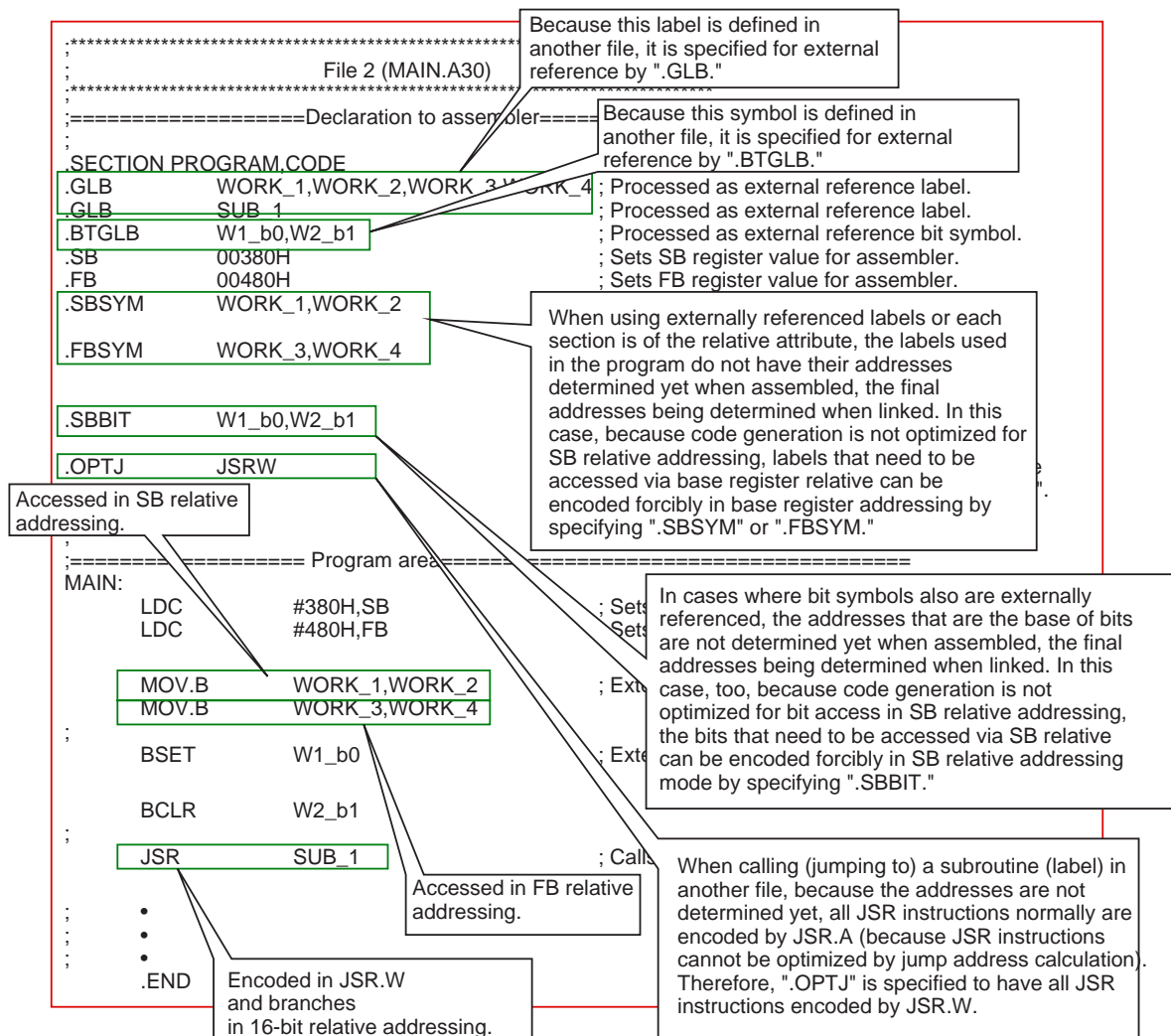


Figure 4.4.3 Divided file 2 (MAIN.A30)

Division Example 3: Subroutine Processing (SUB_1.A30)

Write subroutine processing("PROGRAM" section) and allocated work RAM area("WORK" section) in file 3.

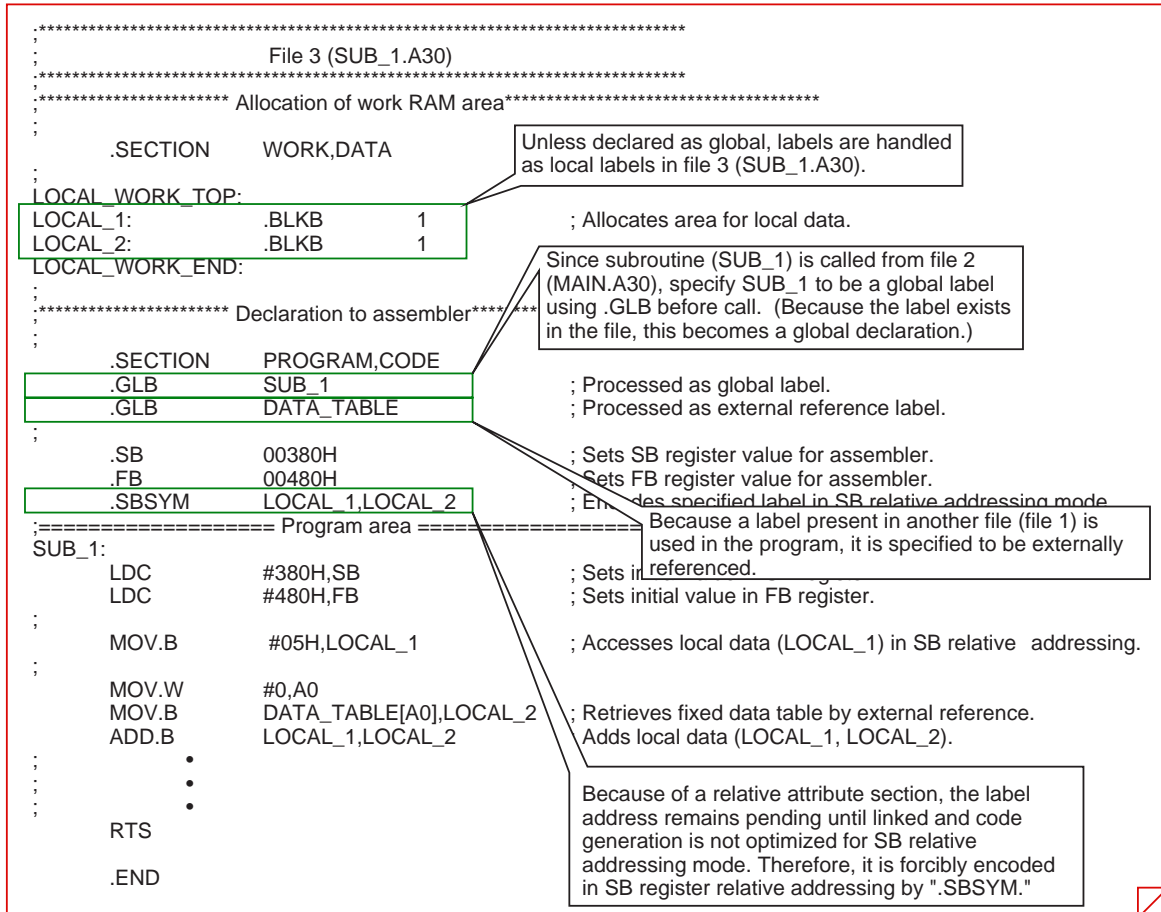
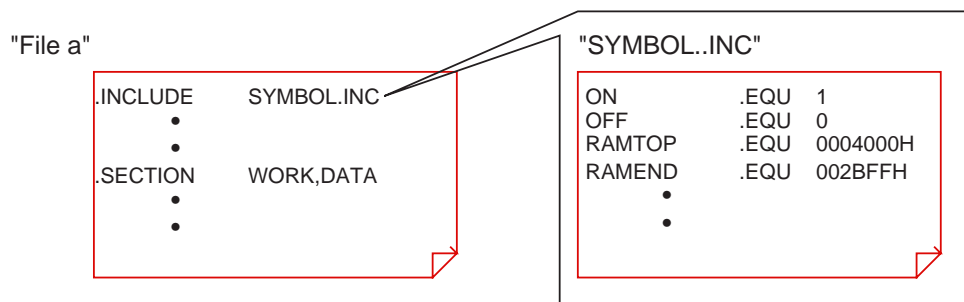


Figure 4.4.4 Divided file 3 (SUB_1.A30)

Making Use of Include File

Normally, write part of external reference specification of symbols and bit symbols (those defined with .EQU, .BTEQU) and/or labels (those having address information) in one include file. In this way, without having to specify external reference in each source file, it is possible to externally reference symbols and labels by reading include files into the source file.

(1) Example for referencing symbols



(2) Example for referencing global labels

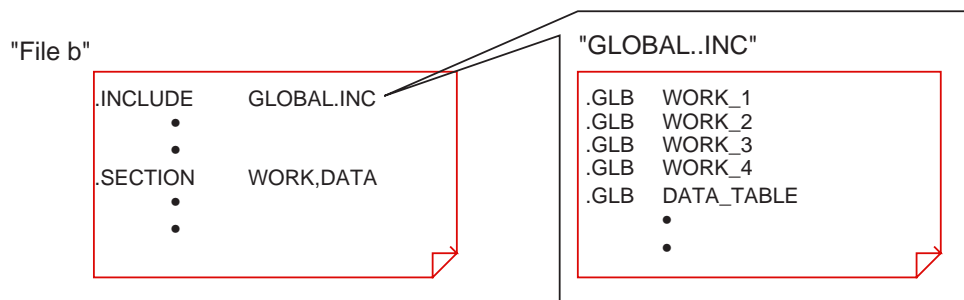


Figure 4.4.5 Example of include file

Making Use of Directive Command .LIST

By writing directive commands ".LIST ON" and ".LIST OFF" at the beginning and end of an include file, it is possible to inhibit the include file from being output to an assembler list file. Figure 4.4.6 shows examples of assembler list files, one not using these directive commands (expansion 1) and one using them (expansion 2).

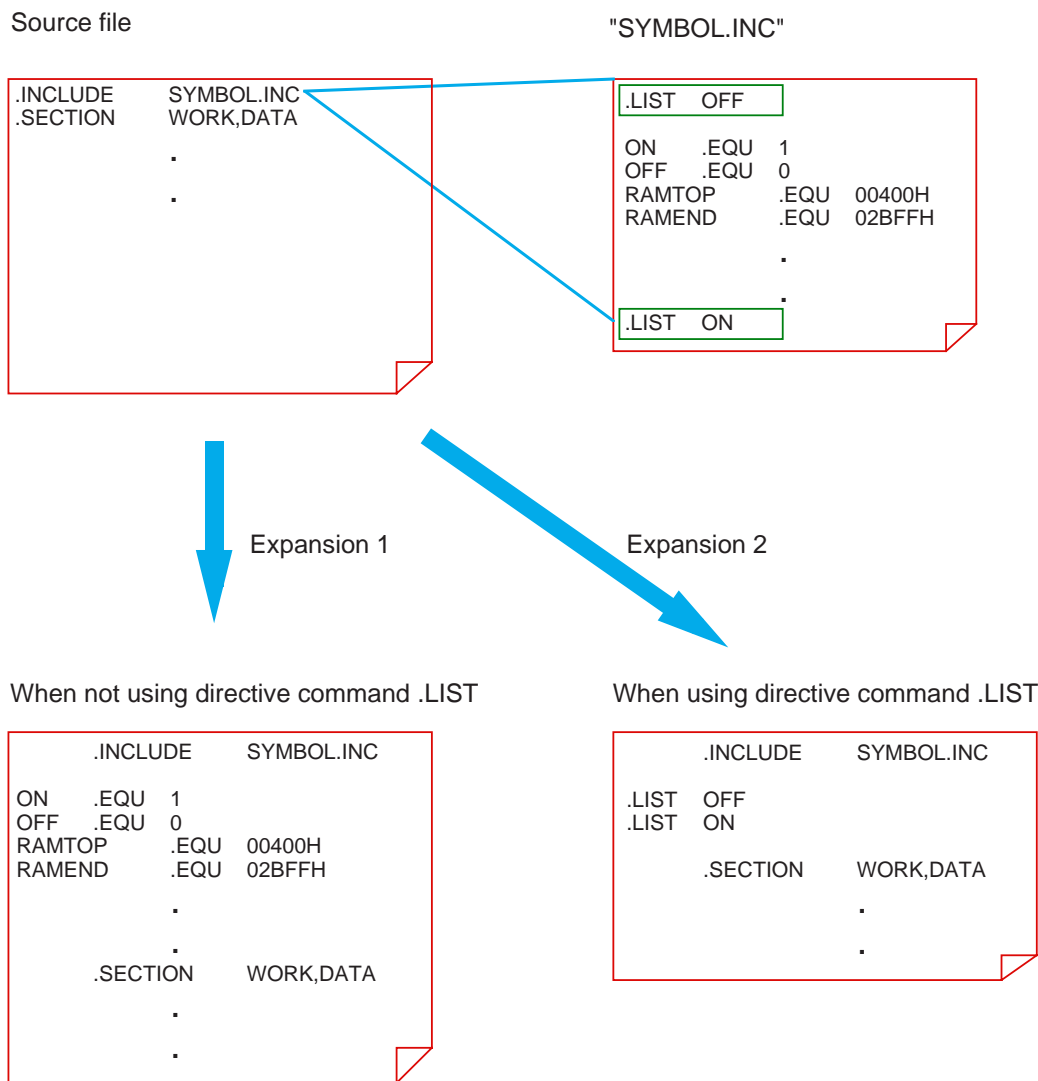


Figure 4.4.6 Utilization of directive command .LIST

4.4.3 Using library files

A library file refers to a collection of several relocatable module files. If there are frequently used modules, collect them in a single library file using the librarian (lib308) that is included with the AS308 system. When linking source files, specify this library file (***.LIB). By so doing, only the necessary modules (those specified in the file as externally referenced) can be extracted when linking. This makes it possible to reduce the assemble time and reuse the program. The following shows an example of how a library file is created and how it is linked.

Creating Library File

Figure 4.4.7 shows an example of how a library file is created.

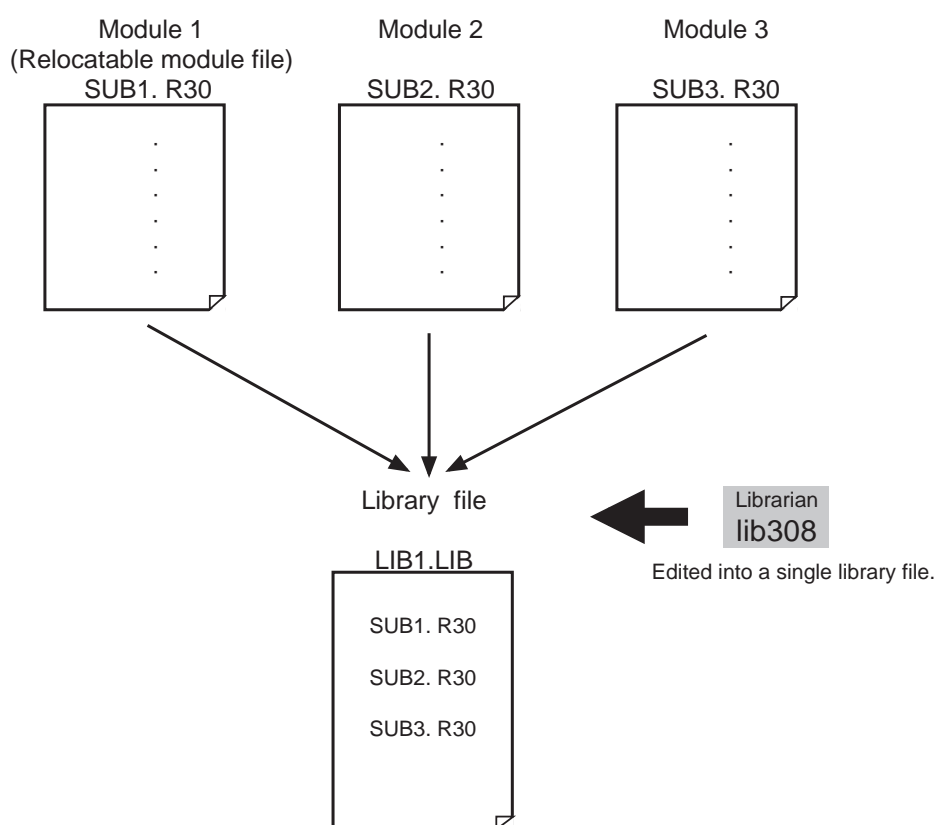


Figure 4.4.7 Creating a library file

Example for Linking Library Files

Figure 4.4.8 shows an example of how library files are linked.

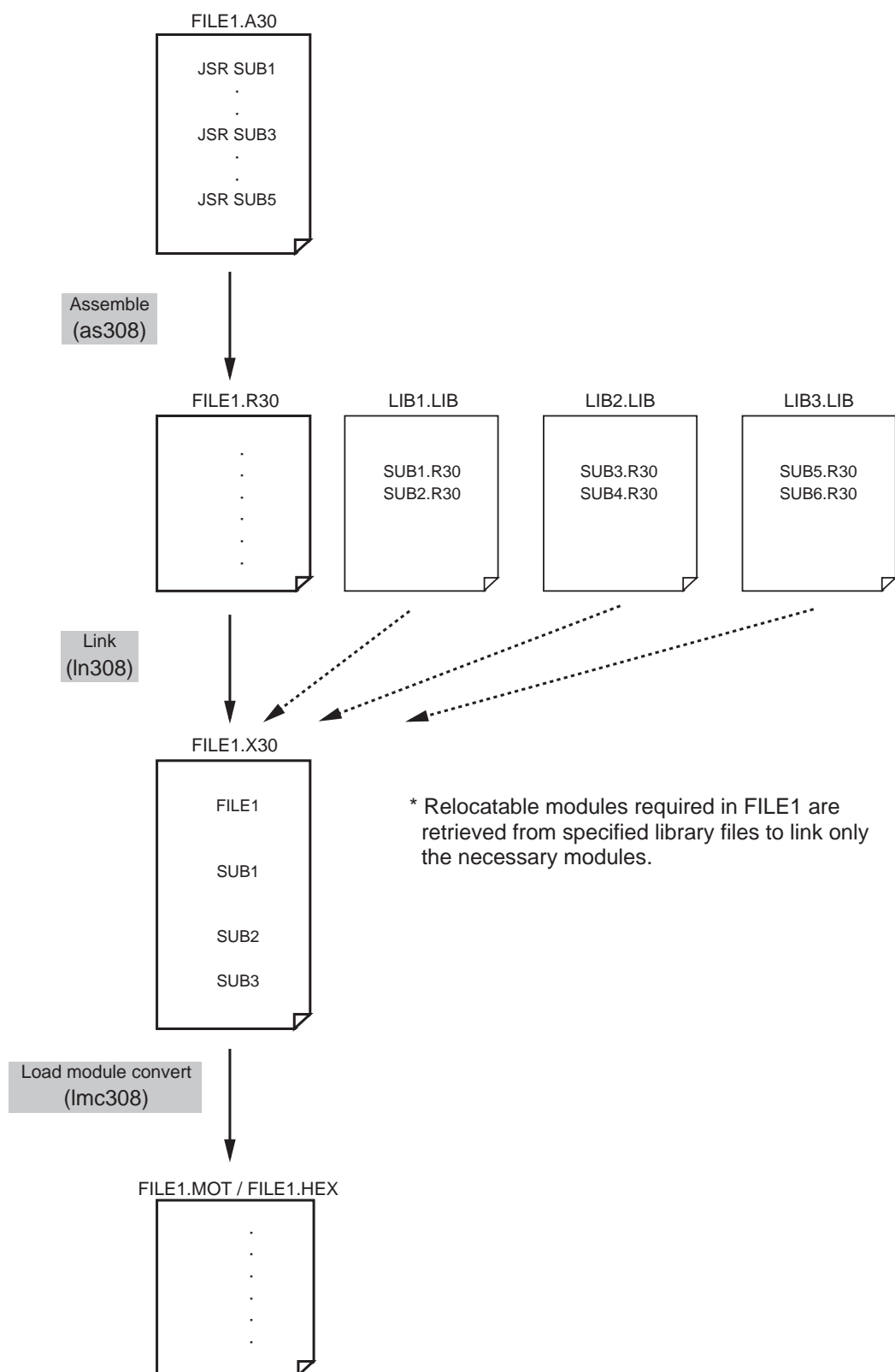


Figure 4.4.8 Example for linking library files and relocatable module file

4.5 A Little Tips...(Programing technique)

This section provides some information, knowledge of which should prove helpful when using the M16C/80 series. This information is provided for several important topics, so refer to the items in interest.

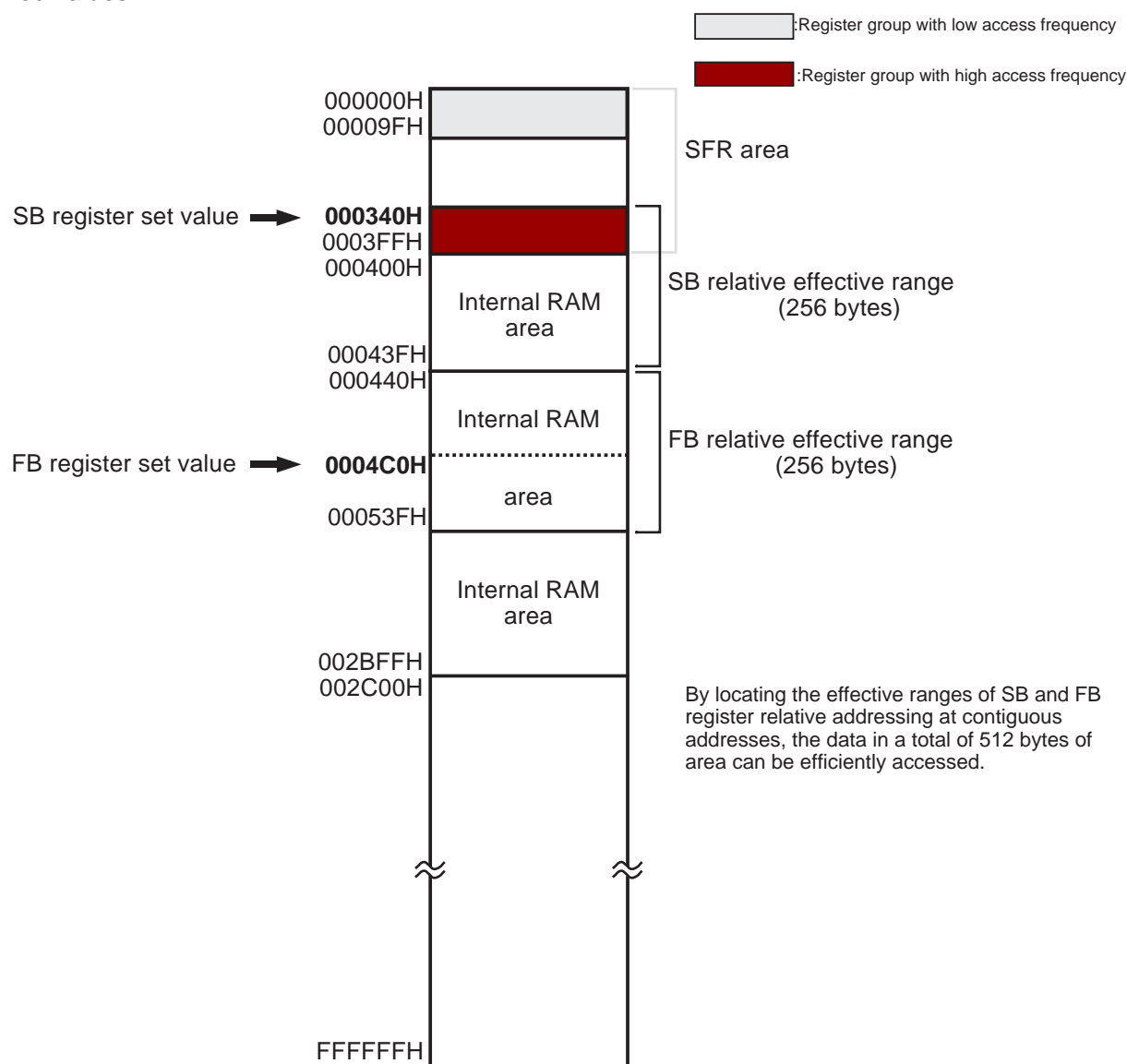
4.5.1 Setup Values of SB and FB Registers

The following explains the setup values of the SB and FB registers.

Basic method for using SB and FB registers

Use the SB and FB registers to set the start address of an area that contains frequently accessed data. Specifically, using these registers to set the frequently used SFR area and work RAM area may prove effective.

Figure 4.5.1 shows an example for setting the SB and FB registers when using them as having fixed values.



Note: Memory map of the M16C/80 group (M30800MC) is used here.

Figure 4.5.1 Example for setting SB and FB registers as having fixed values

Application for using SB and FB registers differently

When using the SB and FB registers after setting them to have fixed values in the program, the address range in which efficient access can be expected is limited to a maximum of 256 bytes each, for a total of 512 bytes.

If use of SB/FB relative addressing over a greater range is desired in order to increase the efficiency of accessing work data or ROM efficiency, the objective may be accomplished by changing the values set in the SB and FB registers "for each subroutine called", in other words by using the registers dynamically.

For an example of how to use, refer to Figure 4.5.3, "Program example for using SB and FB registers dynamically."

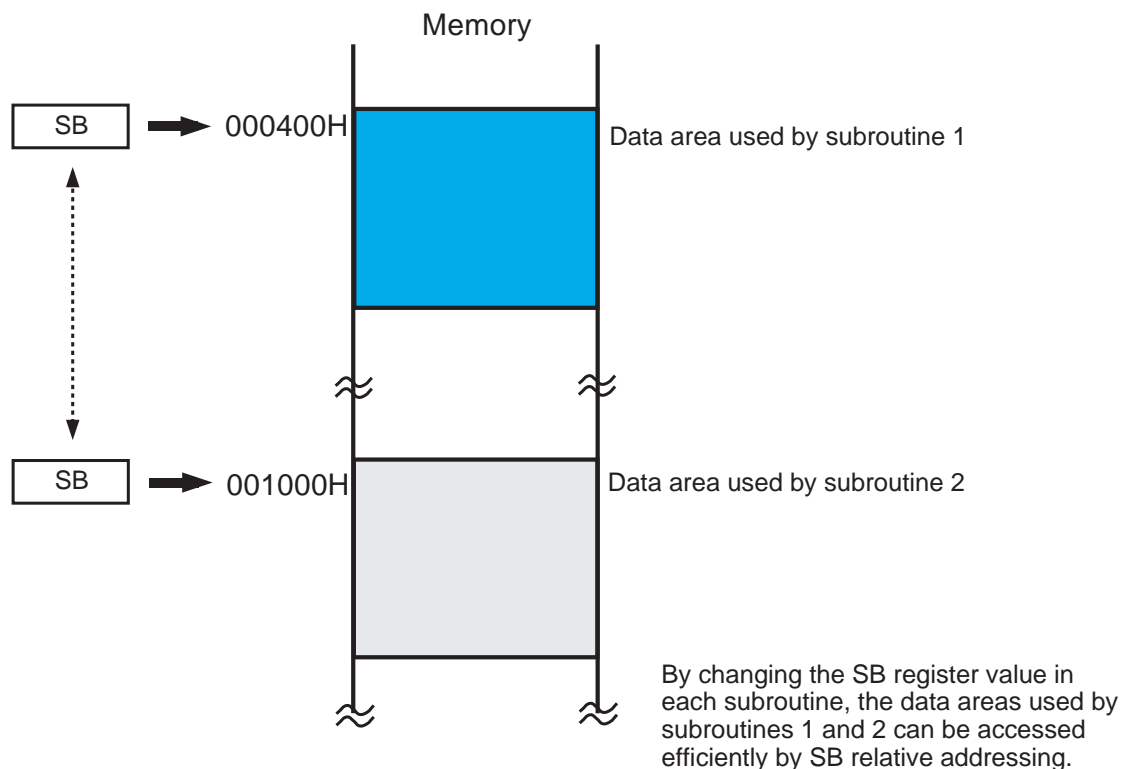


Figure 4.5.2 When using SB and FB registers dynamically

Programming example when using SB and FB registers dynamically

The following shows a program example in which the SB and FB registers are used dynamically.

```

*****Program area *****
;
;===== Start up =====
;
;      .SECTION      PROGRAM,CODE      ; Declares secyion name and section type.
;      .ORG          ROM_TOP           ; Declares start address.
START:
;      LDC          #RAM_END+1,ISP      ; Sets initial value of stack pointer (SP).
;
;      .
;      .
;      .
;
;      .SB          340H                ; Declares SB register value to the assembler.
;      .FB          4C0H                ; Declares FB register value to the assembler.
;      LDC          #340H,SB            ; Sets initial value for SB register.
;      LDC          #4C0H,FB            ; Sets initial value for SB register.
;
;      .
;      .
;      .
;
;=====Main program =====
MAIN:
;      JSR          INIT                ; Initialization routine
;      FSET         I                  ; Enable interrupts
MAIN_10:
;      JSR          SUB_1               ; Calls subroutine "SUB_1"
;
;      .
;      .
;      JSR          SUB_2               ; Calls subroutine "SUB_2"
;
;      .
;      JMP         MAIN_10
;
;===== INIT routine =====
INIT:
;      MOV.B        #0FFH,WORK_1
;      MOV.B        #0FFH,WORK_2
;
;      MOV.B        #01000000B,TA0MR    ; Sets timer A0 mode register.
;      MOV.W        #2500-1,TA0         ; Sets timer A0 count value.
;      MOV.B        #00000111B,TA0IC    ; Sets timer A0 interrupt priority level.
;      BSET         TA0S                ; Timer A0 starts counting.
INIT_END:
;      RTS
;

```

Always be sure to set the same values in the assembler as those set in SB and FB registers.

Set initial values for the SB and FB registers. In the sample program, SB relative addressing can be used in the range of 340H to 43FH, and FB relative addressing in the range of 440H to 53FH, by the main and the INIT routine.

```

===== SUB_1 routine =====
SUB_1:
    .SB    400H    ; Declares SB register value to be changed the assembler.
    .FB    580H    ; Declares FB register value to be changed the assembler.
    LDC     #400H,SB ; Changes SB register value.
    LDC     #580H,FB ; Changes FB register value.
;
;
    MOV.B   WORK_1,R0L
    INC.B   R0L
;
;
;
SUB_1_END:
    RTS
;

===== SUB_2 routine =====
SUB_2:
    .SB    600H    ; Declares SB register value to be changed the assembler.
    .FB    780H    ; Declares FB register value to be changed the assembler.
    LDC     #600H,SB ; Changes SB register value.
    LDC     #780H,FB ; Changes FB register value.
;
;
    MOV.B   WORK_2,R1L
    DEC.B   R1L
;
;
;
SUB_2_END:
    RTS
;

===== Interrupt =====
INT_TA0:
    .SB    1000H    ; Declares SB register value to be changed the assembler.
;
    FSET     B      ; Saves registers( including SB and FB registers)
    LDC     #1000H,SB ; Changes SB register value.
;
;
    MOV.B   #0, COUNT
    DADD.B  #2, DATA
;
;
;
INT_TA0_END:
    REIT
;
;
;
.END

```

Always be sure to specify the values to be changed with ".SB" and ".FB" in the assembler too.

Set the SB and FB register values according to the range of SB and FB relative addressing to be used in the subroutine "SUB_1."
Note: Always be sure to set the SB and FB register values before accessing the work RAM, etc. (Normally set at the beginning of each routine.)

Set the SB and FB register values according to the range of SB and FB relative addressing to be used in the subroutine "SUB_2."
Note: Always be sure to set the SB and FB register values before accessing the work RAM, etc. (Normally set at the beginning of each routine.)

Because interrupts come in asynchronously, always be sure to save the SB and FB register values used by the main routine before setting values back again.

Set the SB and FB register values according to the range of SB and FB relative addressing to be used in the interrupt handler "INT_TA0."

Figure 4.5.3 Program example for using SB and FB registers dynamically

4.5.2 Specifying ROM/RAM data alignments

This section explains how to specify data alignments.

About data alignments

This refers to address adjustment so that when the directive command ".ALIGN" is written, the code in the immediately following line is stored in an even address. For section types "CODE" or "ROMDATA," NOT instructions are written in locations that have been left blank as a result of address adjustment. For section type "DATA," addresses are only adjusted, leaving blank locations intact. If the location where this directive command is written happens to be an even address, no address adjustment is performed.

This directive command can be written in a section that falls under the following conditions:

(1) Relative attribute section for which address adjustment is specified in section definition

```
.SECTION      WORK, DATA, ALIGN
```

(2) Absolute attribute section (no specific limitations)

```
.SECTION      WORK, DATA  
.ORG          400H
```

Advantages of Alignment Specification (Correction to Even Address)

If data of different sizes such as a data table are located at contiguous addresses, the data next to an odd size of data is located at an odd address. In the M16C/80 series, word data (2-byte data) beginning with an even address is read/written in one access, those beginning with an odd address requires two accesses for read/write. Consequently, for data in size of 2 bytes or more such as words and long words, access efficiency and instruction execution speed can be increased by locating data at even addresses. In this case, however, ROM (or RAM) efficiency decreases. Figure 4.5.4 shows an example of a program description that contains alignment specification.

(1) For relative attribute sections

		Address	Code
.SECTION WORK, DATA, ALIGN			
WORK_1 .BLKW	1	00000H	
WORK_2 .BLKW	1	00002H	
WORK_3 .BLKB	1	00004H	
.ALIGN		00005H	Address is incremented by 1.
;			
.			
.SECTION CONST, ROMDATA, ALIGN			
.BYTE	12H	00000H	12H
.ALIGN		00001H	04H NOP code is inserted
.WORD	3456H	00002H	5634H
.			
.			

(2) For absolute attribute sections

		Address	Code
.SECTION WORK, DATA			
.ORG	400H		
WORK_1 .BLKB	1	00400H	
.ALIGN		00401H	Address is incremented by 1.
WORK_2 .BLKW	1	00402H	
WORK_3 .BLKA	1	00404H	
.ALIGN		00407H	Address is incremented by 1.
WORK_4 .BLKL	1	00408H	
;			
.SECTION PROGRAM, CODE			
.ORG	0F0000H		
MOV.W	#0,R0	F0000H	D900H
.			
.			

Figure 4.5.4 Example of alignment specification

4.5.3 Setting stack pointer

The following explains how to set up stack pointers and how to save and restore to and from the stack area when using an interrupt and a subroutine.

Setting Up Stack Pointers (ISP or USP)

(1) Choosing the stack pointer (ISP or USP) to use

When developing a program in only assembly language, normally use the ISP.

When using both ISP and USP, set the initial value of the U flag to 1 (the USP used). As a result, the stack area identified by "USP" is used on the main routine side, while the stack area identified by "ISP" is used by the peripheral I/O interrupt handler routine(note 1).

This allows the amounts of stack used to be estimated separately for main processing and interrupt handling. This should prove effective when jointly developing a program by separating it into files between two or more people. For details, refer to Section 4.3.7, "ISP and USP."

(2) Set the initial value in the selected stack pointer register.

Because the stack in the M16C/80 group is FILO type^(Note 2), it is recommended that the stack pointer initial value be set at the last address of the RAM area.

Also, when registers are saved and restored to and from the stack, the stack pointer changes by 2 at a time when either increased or decreased Therefore, make sure the initial value is always set at an even address. For details, refer to "Saving and restoring to and from the stack" in the next page.

Set up example:

When setting "2C00H" for the interrupt stack pointer (SIP) and "2900H" for the user stack pointer (USP)^(Note 3)

```

;-----Initializing stack pointers-----
;
    LDC      #002C00H,ISP    ; Sets "2C00H" for ISP.
    FSET     U
    LDC      #002900H,SP     ; Sets "2900H" for USP.
    FCLR     U               ; Uses USP on the main side,
                             ; and uses ISP on the interrupt handler routine side.
;

```

Note 1: When using both ISP and USP, be careful not to locate one stack area overlapping the other when allocating memory for the stack. Also, be sure to set values for both stack pointers.

Note 2: FILO (First In, Last Out): When saving registers, they are stacked one on top of another in order of addresses, from large address toward smaller addresses. When restoring registers, they are removed from the stack in the direction toward larger addresses beginning with the last register saved.

Note 3: Because ISP, USP, and FLG are dedicated registers, use the LDC and FSET/FCLR instructions to set these registers.

Saving and restoring to and from the stack

Registers, etc. are saved and restored to and from the stack in the following cases:

(1) When an interrupt is accepted

When an interrupt is accepted, the registers shown below are saved to the stack area:

Program Counter (PC) -> 4 bytes (The most significant byte is fixed to 00H.)

Flag Register (FLG) -> 2 bytes ... 6 bytes in total

However, if the accepted interrupt is a fast interrupt, the Flag Register (FLG) and Program Counter (PC) respectively are saved to the Save Flag Register (SVF) and Save PC Register (SVP), with nothing placed in the stack.

After interrupt handling is finished, the above saved registers are restored from the stack area by the REIT instruction.

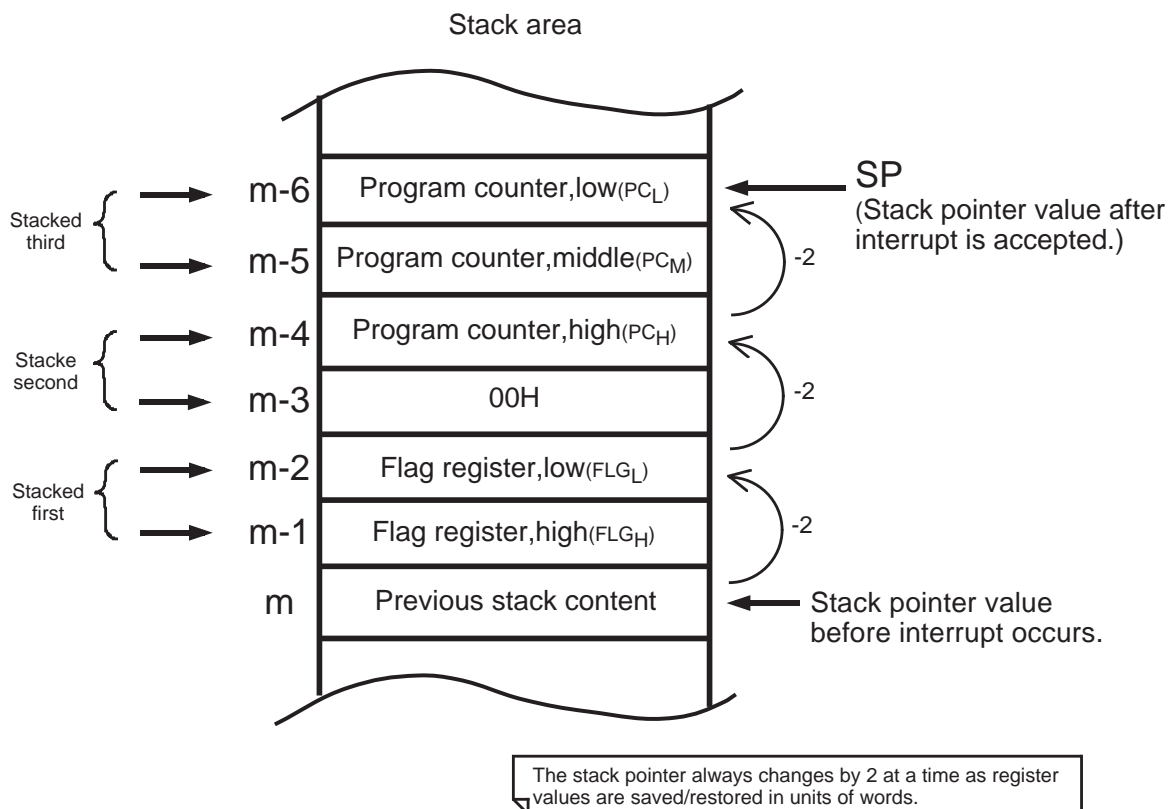


Figure 4.5.5 Stack operation and status when an interrupt is accepted

Note: Even when one byte of data are saved/restored using push and pop instructions (e.g., PUSH, POP, PUSHM, and POPM), the stack pointer always changes by 2 at a time.

(2) When calling a subroutine (when executing JSR, JSRI, or JSRS instruction)

When the JSR, JSRI, or JSRS instruction is executed, the register shown below is saved to the stack area:

Program Counter (PC) -> 4 bytes (The most significant byte is fixed to 00H.)

When the subroutine is completed, the above saved register is restored from the stack area by the RTS instruction.

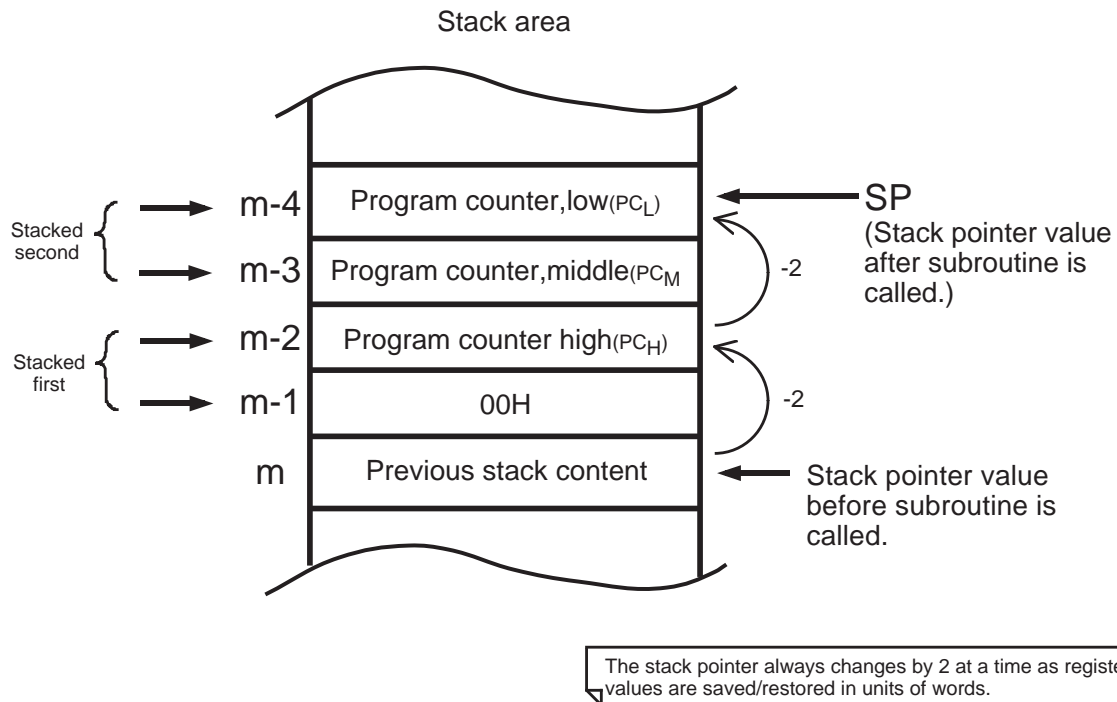


Figure 4.5.6 Stack operation and status when a subroutine is called

The M16C/80 series has a reserved area in the fixed vectors that is called the "special page vector table," with each vector assigned a special page number. (Refer to Section 2.1.3, "Fixed Vector Area.") This vector table can be used to store subroutine or jump addresses, and by specifying a special page number in the special page subroutine call instruction (JSRS) or special page jump instruction (JMPS), it is possible to branch off in fewer bytes than possible with the ordinary subroutine call instruction (JSR) or jump instruction (JMP)^(Note). As a result, the number of program steps and the ROM size can be reduced.

Shown below is a program example for subroutine call using a special page.

The diagram illustrates assembly code segments with callouts explaining specific instructions:

```

*****Program area *****
===== Startup =====
;
;
;
;SECTION      PROGRAM CODE          ; Program section
;.ORG        0FF0000H                ; Specifies program location address.
START:
LDC          #RAM_END+1,ISP           ; Sets initial for stack pointer(ISP).
;
;
;
;----- Main program -----
MAIN:
MAIN_10:    JSR      INIT              Called by ordinary subroutine call instruction
            JSRS     #255              Called by specifying the special page vector number (No. 255) that contains the start address of subroutine "SBU_1."
                                           ; Calls subroutine "SUB_1" using special page subroutine call.
;
;
;
;----- INIT routine -----
INIT:
MOV.B #0FFH,WORK_1
MOV.B #0FFH,WORK_2
;
;
;
INIT_END:
RTS
;

```

Callout 1: Because the most significant byte of the jump address in special page subroutine call is fixed to "FFH," the program (subroutine) must be located at addresses FF0000H to FFFFFFFH.

Callout 2: Called by ordinary subroutine call instruction

Callout 3: Called by specifying the special page vector number (No. 255) that contains the start address of subroutine "SBU_1."

Callout 4: ; Calls subroutine "SUB_1" using special page subroutine call.

Callout 5: ; Calls subroutine "SUB_1" using special page subroutine call.

Callout 6: Called by specifying the special page vector number (No. 251) that contains the start address of subroutine "SBU_2."

Note: If the branch distance specifier ".S" or ".B" is used, code size is smaller for ordinary jump instructions than for special page jump.

4.5 A Little Tips...(Programing technique)

```

===== SUB_1 routine =====
SUB_1:      MOV.B   WORK_1,R0L
            INC.B   R0L
;
;          .
;          .
SUB_1_END:  RTS
;
===== SUB_2 routine =====
SUB_2:      MOV.B   WORK_2,R1L
            DEC.B   R1L
;
;          .
;          .
SUB_2_END:  RTS
;
***** Setting fixed vector (Special page vector and fixed interrupt vector) *****

.SECTION    F_VECT,ROM,DATA
.ORG        0FFFE00H                ; Specifies address of special page vector No.255
                                         ; (start address of fixed vector area).
.WORD       SUB_1 & 00FFFFFH        ; Sets start address of subroutine "SUB_1".

.ORG        0FFFE08H                ; Specifies address of special page vector No.251.
.WORD       SUB_2 & 00FFFFFH        ; Sets start address of subroutine "SUB_1".

.ORG        0FFFDDCH                ; Start address of fixed interrupt vector
.LWORD      dummy
.LWORD      dummy
.LWORD      dummy
.LWORD      dummy
.LWORD      dummy
.LWORD      dummy
.LWORD      dummy
.LWORD      dummy
.LWORD      START
.END

```

When called by a special page, because the most significant byte (bits 16 to 23) of the jump address is fixed to "FFH," the subroutine must be located within addresses FF0000H to FFFFFFFH.

Because labels are expanded into 3-byte quantities by the assembler, mask the most significant byte of address with the assembler operator "&" (logical AND) and set the 2 low-order bytes of address in the special page vector.

Because special page vectors are comprised of 2 bytes each, special page vector No. 251 is at address FFFE08H.

Figure 4.5.7 Example for using special page subroutine call

4.5.5 Example for using software interrupt (INTO instruction)

The INTO instruction (overflow interrupt) is a software interrupt instruction that generates an interrupt when it is executed while the Flag Register (FLG)'s overflow flag (O) is set to 1.

Therefore, the INTO instruction can be used to call an overflow handling routine when the operation of a divide instruction (e.g., DIV, DIVU) or multiply/accumulate instruction (RMPA) resulted in an overflow.

Figure 4.5.8 shows an example of how to use the INTO instruction.

Example for using the INTO instruction

```

*****Program area *****
;=====Main program =====
MAIN:
MAIN_10:   JSR      INIT          ; Initialization routine
          MOV.L     DATA1, R2R0
          DIV.W     #5           ; Signed divide
          INTO      ; Overflow interrupt
          MOV.W     R0, ANS_DAT1
          ;
          ;
          MOV.L     #0, R2R0
          MOV.W     #0, R1
          ;
          MOV.L     #10000H, A0   ; Sets address in which to store multiplicand.
          MOV.L     #20000H, A1   ; Sets address in which to store multiplier.
          MOV.W     #0FFH, R3     ; Sets number of times products are summed.
          RMPA.W    ; Perform multiply / accumulate operation
          INTO      ; Overflow interrupt
          ;
          MOV.L     R2R0, ANS_DAT2
          MOV.W     R1, ANS_DAT2+4
          ;
          ;
          JMP      MAIN_10
          ;
;===== INIT routine =====
INIT:
          MOV.W     #0FFFFH, DATA1
          MOV.W     #0, ANS_DAT1
          ;
          ;
          ;
INIT_END:  RTS
          ;

```

Only when the operation resulted in an overflow (O flag = 1), an interrupt is generated by the INTO instruction; otherwise, no interrupt is generated and the next instruction is executed.

When an overflow occurs during the multiply/accumulate operation (O flag = 1), the instruction being executed is suspended and the next instruction (INTO instruction) is executed, so that an interrupt is generated by the INTO instruction.

```

INT_OVER_FLOW:
    FSET        B
;
;      Processing performed when the operation resulted in an overflow
;
INT_OVER_FLOW_END:
    REIT
;
;===== Dummy interrupt program =====
dummy:
    REIT
;
;***** Setting fixed vector *****
;
    .SECTION      F_VECT,ROMDATA
    .ORG          0FFFFDCH
    .LWORD        dummy
    .LWORD        INT_OVER_FLOW
    .LWORD        dummy
    .LWORD        dummy
    .LWORD        dummy
    .LWORD        dummy
    .LWORD        dummy
    .LWORD        dummy
    .LWORD        START
;
    .END

```

Notice that the vector for the overflow interrupt (INTO instruction) is a fixed vector.

; Start address of fixed interrupt vector
; Undefined instruction interrupt vector
; Sets start address of interrupt handler
; for Overflow(INT O instruction) interrupt vector
; BRK instruction interrupt vector
; Address match interrupt vector
; Unused
; Watchdog timer interrupt vector
; Unused
; NMI interrupt vector
; Sets reset vector

Figure 4.5.8 Example for using INTO (software interrupt) instruction

4.5.6 Software runaway prevention

This section explains how to prevent the program from going wild by means of software, for example, using a watchdog timer or software interrupt instruction.

Using a watchdog timer

The watchdog timer is a 15-bit timer, which is used to detect occurrence of program runaway. When the program goes wild, the watchdog timer under-flows, generating an interrupt. The program can be restarted in this watchdog timer interrupt handling by, for example, a software reset.

The watchdog timer interrupt is non-maskable. After a reset, the watchdog timer remains idle, and is made to start counting by a write to the watchdog timer start register. Note that the watchdog timer is initialized when the CPU is reset, when data is written to the watchdog timer start register, and when a watchdog timer interrupt request is generated.

Method for Detecting Program Runaway

The chart below shows an operation flow when the program is found out of control and the method of runaway detection.

(1) Operation flow

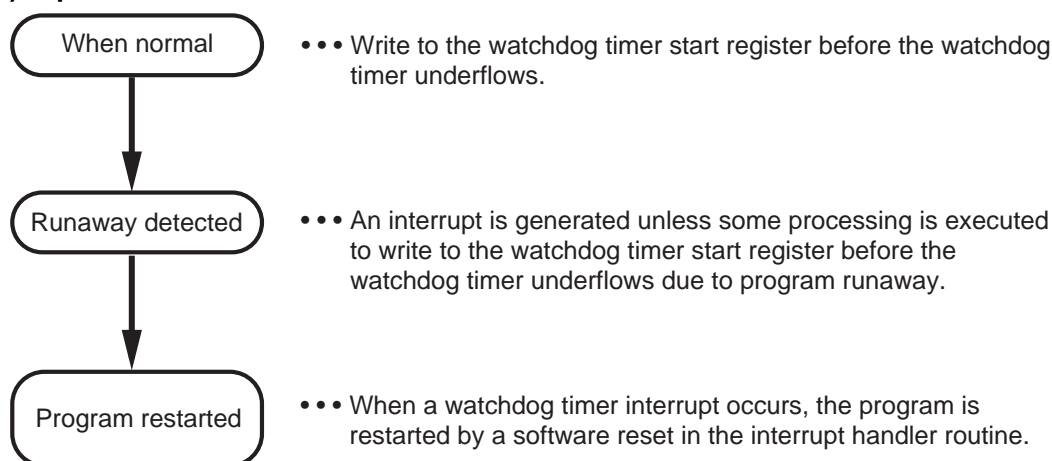


Figure 4.5.9 Operation flow when program runaway is detected

(2) Method of runaway detection

Program a procedure so that a write to the watchdog timer start register is performed before the watchdog timer under-flows. By writing to the watchdog timer start register, the initial count "7FFFH" is set in the watchdog timer. (This is fixed, and not other value can be set.)

If this write operation is inserted in a number of locations, it can happen that a write to the watchdog timer start register is performed at a place to which the program has been brought by runaway. Thus, no where in the program can it be detected to have run out of control.

Therefore, be careful that this write operation is inserted in only one location such as the main routine that is always executed. However, consider the length of the main routine and that of the interrupt handler routine to ensure that a write to the watchdog timer start register will be performed before a watchdog timer interrupt occurs.

(3) Restarting the program after having gone wild

Write your program so that Processor Mode Register 0 bit 3 (software reset bit) is set by writing a 1 in an interrupt handler routine. This generates a software reset, so the program restarts from its reset state. (The internal RAM contents retained at this time are those that were being held immediately before the reset.)

Make sure the start address of this interrupt handler routine is set in the interrupt vector for the watchdog timer interrupt beforehand.

When restarting the program from its reset state, always be sure to use the software reset bit to reset it. Note that if the address value that has been set in the interrupt vector for the watchdog timer interrupt is the same as that of the reset vector, the IPL (processor interrupt priority level) remains 7 without being cleared. Therefore, when the program restarts, a problem is encountered that all other interrupts are disabled.

Examples of Runaway Detection Programs

Figures 4.5.10 and 4.5.11 show sample programs in which the watchdog timer is used to detect program runaway.

Example 1: Operation (subroutine) for writing to the watchdog timer start register is executed periodically at predetermined intervals

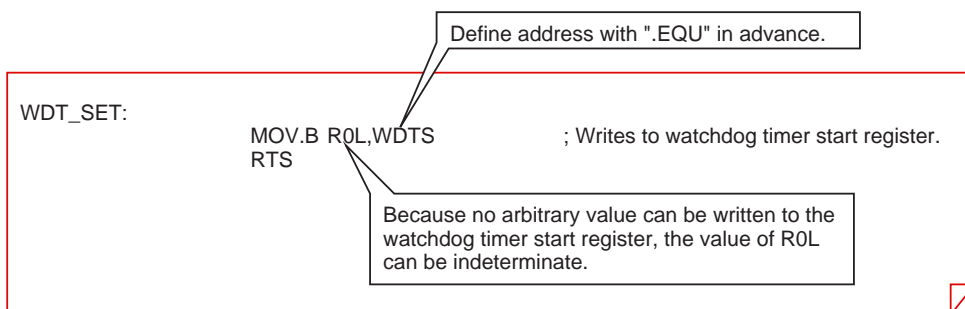


Figure 4.5.10 Example of runaway detection program 1

Example 2: Interrupt handling program to restart the system is executed when a watchdog timer interrupt occurs

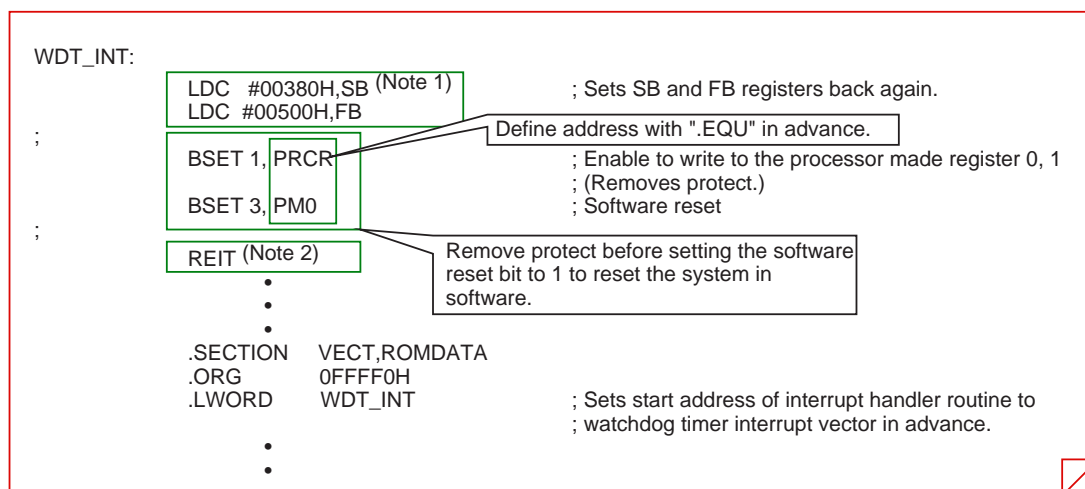


Figure 4.5.11 Example of runaway detection program 2

Note 1: If the program runs out of control, the contents of the base registers (SB, FB) are not guaranteed. Therefore, they must be set correctly again before writing values to the SFR.

Note 2: The system enters a reset sequence immediately after the software reset bit is set to 1. Therefore, no instructions following it are executed.

Using software interrupts (UND/BRK instructions)

Both BRK and UND instructions are software interrupt instructions that generate an interrupt when the instruction is executed. These instructions can also be used to detect occurrence of program runaway. The following shows how to detect.

The method of detecting runaway

Program runaway detection can be accomplished by embedding the BRK or UND instruction in an area of ROM other than one being used as the program area beforehand. When the program goes wild and accesses an unused area of ROM, it fetches the UND or BRK instruction stored in the area, at which time an interrupt is generated, providing a means of detecting runaway.

Also, by storing the start address of a dummy interrupt handler in an unused interrupt vector beforehand, it is possible to prevent the program from going wild in the event an unused interrupt occurs. For description examples, refer to Section 4.3.6, "Sample List 3 (Using Interrupts)."

To restart the program that has gone wild and handle the generated interrupt, follow the same procedure as when using the watchdog timer that is described earlier.

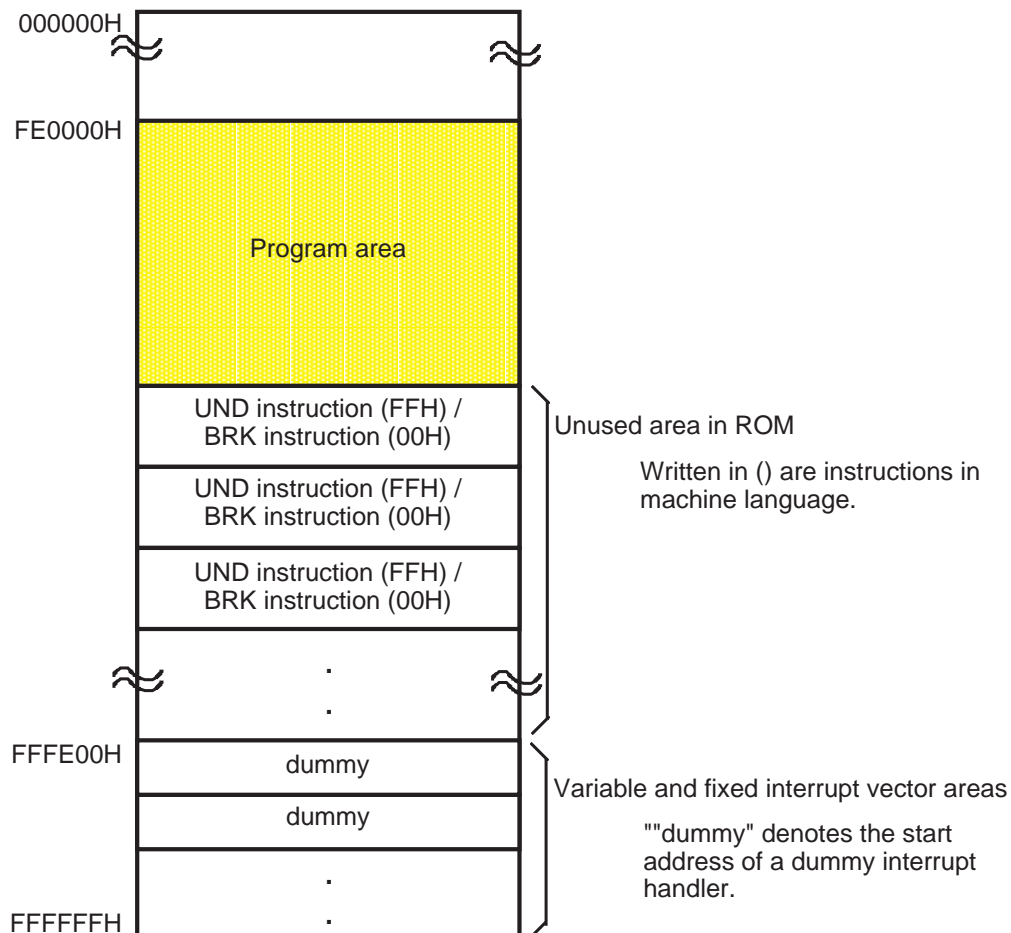


Figure 4.5.12 Runaway detection using software interrupt instructions

4.5.7 Method for using the "-LOC" option

This section explains how to use the linker (LN308) option "-LOC" (specify section data location) that is included in the M16C/80 series assembler system.

About the "-LOC" option

The "-LOC" option specifies the address at which to store the internal data of a specified section, and is used when modules need to be stored in other areas than the run-time storage area(note). Therefore, the internal label values (address values) of a specified section are generated with respect to the address specified by ".ORG" in the source file or the address specified by the linker option "-ORDER" when linking.

Usage example

Shown below is an example where the section name "PROGRAM" that is stored at address EF0000H is transferred to address 1000H before program execution, then the program is executed from address 1000H.

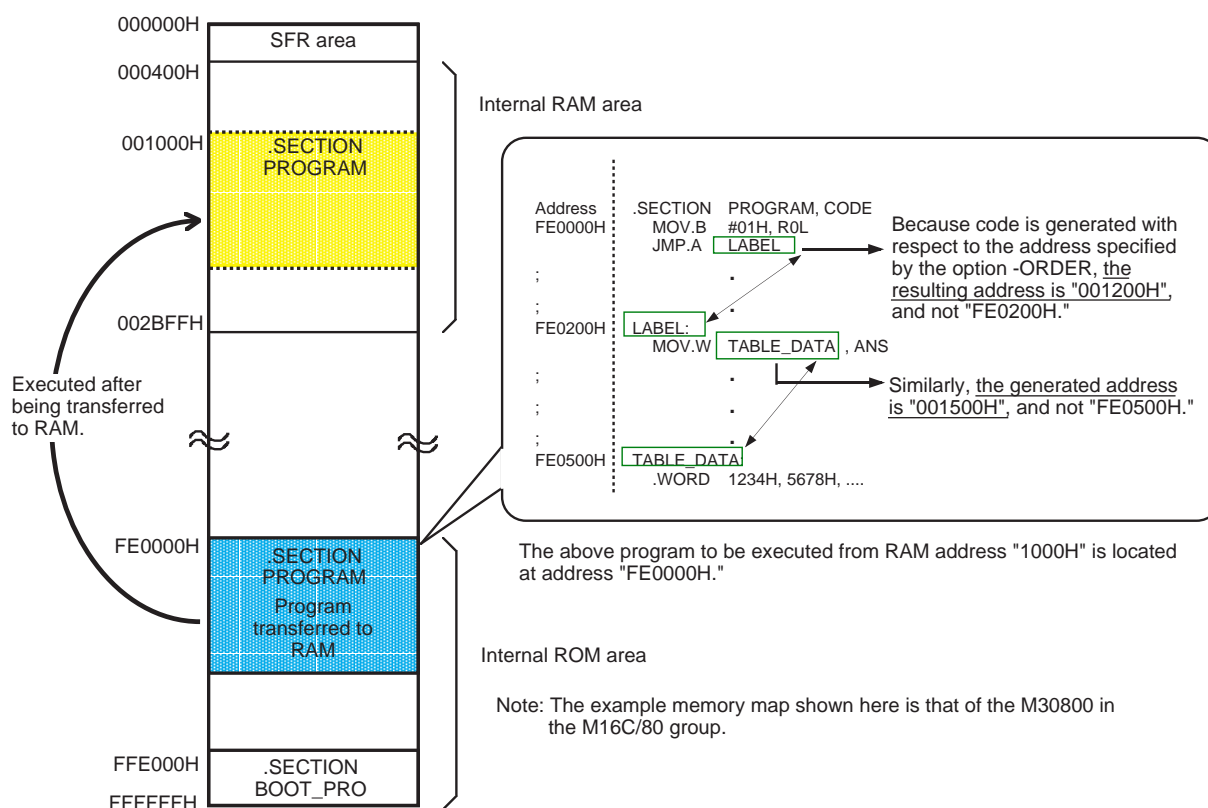


Figure 4.5.13 Example for specifying section data location with -LOC option

Note: This method may be used for the flash ROM version of the M16C/80 series where when writing a program to the internal flash ROM in CPU rewrite mode, the program used to write to the flash ROM is run in RAM.

4.6 Standard processing program

This section shows examples of commonly used processing in programming of the M16C/80 series. For more information, refer to Application Notes, "M16C/80 Series Sample Programs Collection".

Conditional Branching Based on Specified Bit Status

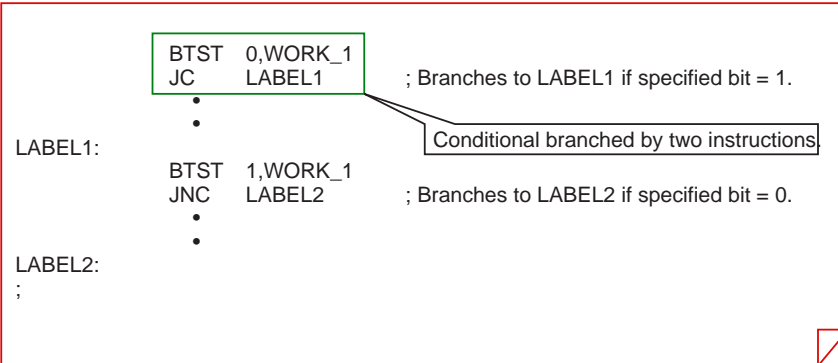


Figure 4.6.1 Sample program for conditional branching based on specified bit status

Retrieving Data Table

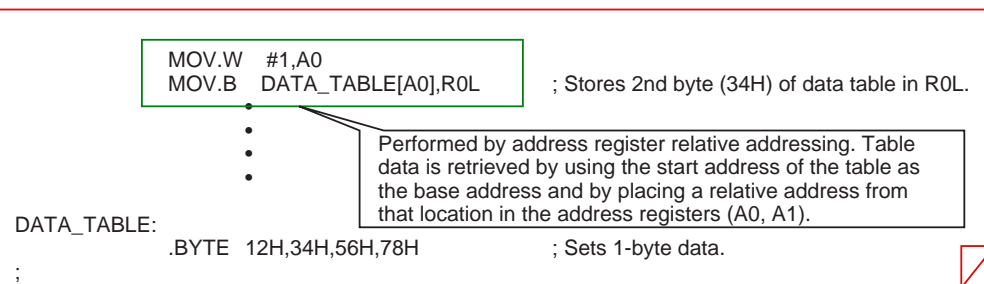


Figure 4.6.2 Sample program for table retrieval

Subroutine call by table jump

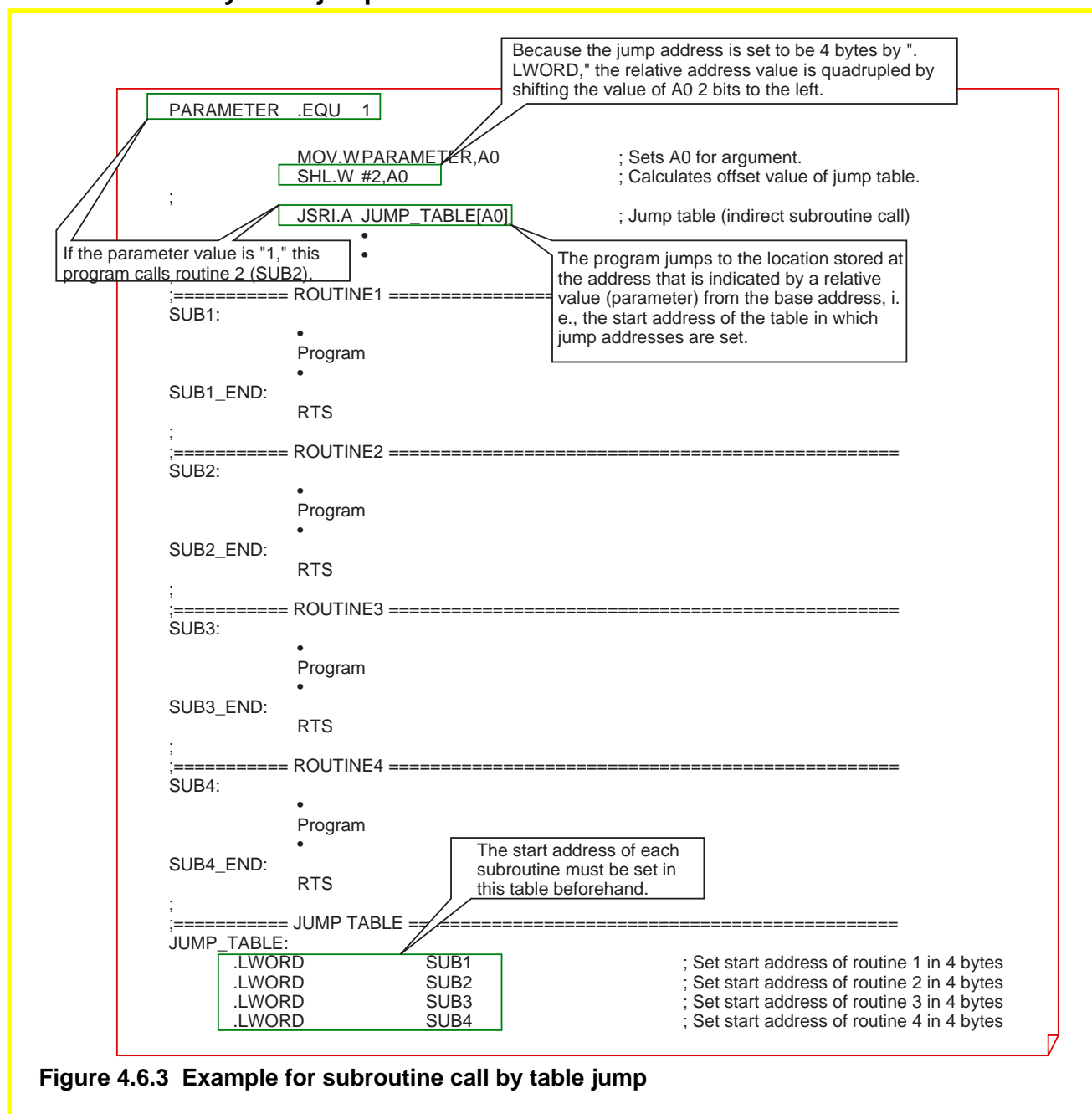


Figure 4.6.3 Example for subroutine call by table jump

Appendix

Command input form and command parameters in AS308 system

Appendix A. Generating Object Files

A-1 Assembling (as308)

A-2 Link (ln308)

A-3 Generating
Machine Language File

Appendix A Generating Object Files

The AS308 system is a program development support tool consisting of an assembler (as308), linkage editor (ln308), load module converter (lmc308), and other tools (lb308, abs308, and xrf308). This section explains how to generate object files using the AS308 system.

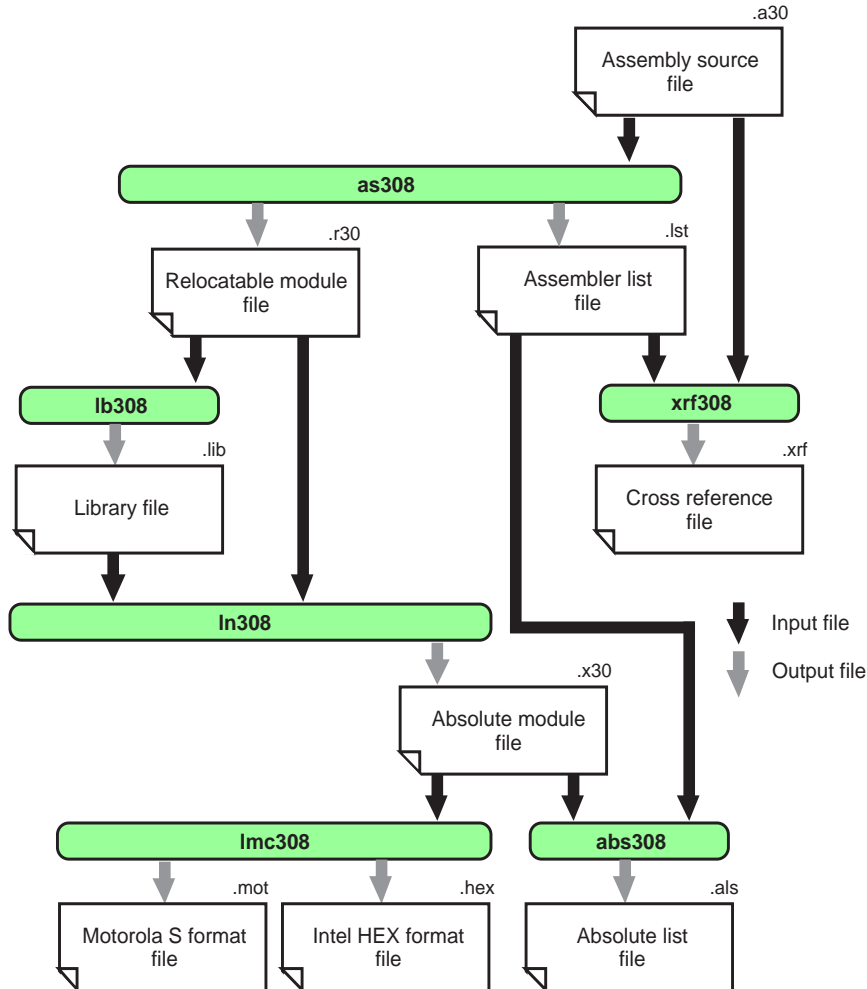


Figure A.1 Outline of processing by AS308

Note: In this manual, the AS308 system is referred to by "AS308 system" (uppercase) when it means the entire system or by "as308" (lowercase) when it means only the assembler (as308).

Appendix A-1 Assembling (as308)

The following explains the files generated by the relocatable assembler (as308) and how to start up the assembler.

Files Generated by as308

(1) Relocatable module file (*.R30) ... Generated as necessary**

This file is based on IEEE-695. It contains machine language data and its relocation information.

(2) Assembler list file (*.LST) ... Generated when option '-L' is specified**

This file contains list lines, location information, object code, and line information. It is used to output these pieces of information to a printer.

(3) Assembler error tag file (*.TAG) ... Generated when option '-T' is specified**

This file contains error messages for errors that occurred when assembling the source file. This file is not generated when no error was encountered. This file allows errors to be corrected easily when it is used in an editor that has the tag jump function.

Method for Starting Up as308

>as308 file name.extension [file name.extension...] [option]

Be sure to write at least one file name. The extension (.A30) can be omitted.

Table A.1 Command Options of as308

Command option	Function
-.	Inhibits assemble processing messages from being output.
-A	Evaluates mnemonic operand.
-abs16	Specifies 16-bit absolute addressing mode. *Always input this option in small letters.
-C	Displays command options when as308 has started up mac308 and asp308.
-D symbol name=constant	Sets symbol constant.
-F expansion file name	Fixes expansion file of directive command ..FILE.
-H	Header information is not output to an assembler list file.
-I	The include file specified by ".INCLUDE" that is written in the source file is searched from a specified directory.
-L	-L The software generates an assembler list file. -LC Line concatenation is output directory as is to a list file. -LD Information before .DEFINE is replaced is output to a list file. -LI Even program sections in which condition assembler resulted in false conditions are output to the assembler list file. -LM Even macro description expansion sections are output to the assembler list file. -LS Even structured description for AS30 expansion sections are output to the assembler list file.
-mod60	AS308 replaces some of the commands in the program written for AS30. *Always input this option in small letters.
-mod60p	Processes structured commands for AS30. *Always input this option in small letters.
-N	Inhibits line information of macro description from being output to relocatable module file.
-O directory path name	Specifies directory for file generated by assembler. Do not insert space between the letter O and directory name. (Default is current directory.)
-S	Outputs local symbol information to relocatable module file.
-T	Generates tag file.
-V	Displays version of assembler system each program.
-X program name	Generates error tag file and invokes command.

Example for Using as308 Commands

Example:

>as308 -L -Oc:\work SAMPLE

Separate each option with a space.

If extension is omitted, ".A30" is assumed.

This command generates SAMPLE.LST and SAMPLE.R30 from SAMPLE.A30 and outputs them to the \work directory.

Command options can be written in uppercase or lowercase as desired.

>as30 -s -t sample

This command outputs the system label and local symbol information of SAMPLE.A30 to the relocatable module file SAMPLE.R30.

Assembler List File

Figure A.2 shows an example of the assembler list file.

```

* M16C/80 SERIES ASSEMBLER * SOURCE LIST   Wed Mar 24 16:04:39 1999  PAGE 001
SEQ. LOC.  OBJ.  0XMSDA  *...* SOURCE STATEMENT...8...9...0...1...2...3...4
1          ; "Sample List"
2          ; ***** Include *****
3          ;
4          ; .INCLUDE      M30800.INC
5          ; -----
6 1         ;             M30800 SFR defined file
7 1         ; -----
8 1         ; .LIST      OFF
9 1         ; .LIST      ON
10         ;
11         ; ***** Defined symbol *****
12         ;
13 00000400h      RAM_TOP      .EQU  000400H      ; Start address of RAM area
14 00002BFFh      RAM_END      .EQU  002BFFH      ; End address of RAM area
15 00FE0000h      ROM_TOP      .EQU  0FE0000H      ; Start address of ROM area
16 00FFFDCh      FIXED_VECT_TOP .EQU  0FFFDCh      ; Start address of fixed vector
17 00000400h      SB_BASE      .EQU  000400H      ; Base address for SB relative
18 00000580h      FB_BASE      .EQU  000580H      ; Base address for FB relative
19 00000300h      ISP_SIZE     .EQU  300H         ; Size of interrupt satck area
20         ;
21         ; ***** Allocated work RAM area *****
22         ;
23         ; .SECTION      WORK,DATA
24 000400         .ORG       RAM_TOP
25         ;
26 000400         WORKRAM_TOP:
27 000400(000001H) char:      .BLKB  1             ; Allocates a 1-byte area.
28 000401(000002H) short:     .BLKW  1             ; Allocates a 2-byte area.
29 000403(000003H) addr:      .BLKA  1             ; Allocates a 3-byte area.
30 000406(000004H) long:      .BLKL  1             ; Allocates a 4-byte area.
31 00040A         WORKRAM_END:
32         ;

```

Figure A.2 Example of assembler list file

Appendix

Command input form and command parameters in AS308 system

```

33          ;***** Defined bit symbol *****
34          ;
35          char_b0      .BTEQU      0,char      ; Bit 0 of char
36          short_b1     .BTEQU      1,short     ; Bit 1 of short
37          addr_b2      .BTEQU      2,addr      ; Bit 2 of addr
38          long_b3      .BTEQU      3,long       ; Bit 3 of long
39          ;
* M16C/80 SERIES ASSEMBLER * SOURCE LIST   Wed Mar 24 16:04:39 1999  PAGE 002
SEQ. LOC.  OBJ.      0XMSDA *...*...SOURCE STATEMENT...8...*...9...*...0...*...1...*...2...*...3...*...4
40          .PAGE
41          ;***** Program area *****
42          ;===== Startup =====
43          ;
44          .SECTION     PROGRAM, CODE
45          FE0000       .ORG      ROM_TOP
46          FE0000       START:
47          FE0000 D52F002C00      LDC      #RAM_END+1,ISP      ; Sets initial value for stack pointer(ISP)
48          ;
49          FE0005 F6E30A00 Q      MOV.B    #03H,PRCR          ; Removes protection.
50          FE0009 1504008301 S     MOV.W    #0183H,PM0        ; Sets processor mode register 0 and 1.
51          FE000E 1506000820 S     MOV.W    #2008H,CM0        ; Sets system clock control registers 0 and 1.
52          FE0013 140B0012 S     MOV.B    #12H,MCD           ; Sets main clock divide register.
53          FE0017 120A00 Z      MOV.B    #0,PRCR             ; Protects all registers.
54          ;
* M16C/80 SERIES ASSEMBLER * SOURCE LIST   Wed Mar 24 16:04:39 1999  PAGE 002
SEQ. LOC.  OBJ.      0XMSDA *...*...SOURCE STATEMENT...8...*...9...*...0...*...1...*...2...*...3...*...4
85          ;
86          ;===== MAIN =====
87          FE0066       MAIN:
88          FE0066 B88B00E0FF      MOV.B    DATA_TABLE[R0],R0L
89          FE006B 99EF3412      MOV.W    #1234H,R1
90          FE006F D2B800      BSET      char_b0
91          ;
92          ;
93          ;
94          FE0072 B8AB00E0FF      MOV.B    DATA_TABLE,R0L
95          FE0077 BBEE          JMP      MAIN
96          ;
101         ;=====
102         ;
103         ;
104         FFE000
105         ;
106         FFE000       DATA_TABLE:
107         FFE000 12345678      .BYTE     12H,34H,56H,78H      ; Sets 1 byte data.
108         FFE004 34127856      .WORD     1234H,5678H         ; Sets 2 bytes data.
109         FFE008 563412BC9A78  .ADDR     123456H,789ABCH      ; Sets 3 bytes data.
110         FFE00E 78563412      .LWORD    12345678H,9ABCDEF0H ; Sets 4 bytes data.
111         F0DEBC9A
112         FFE016       DATA_TABLE_END:
113         ;
114         ;
115         ;
116         ;
117         ;
118         .END

```

Information List

TOTAL ERROR(S) 00000	Outputs total number of errors derived from assembling, as well as total number of warnings and total number of list lines.
TOTAL WARNING(S) 00000	
TOTAL LINE(S) 00127 LINES	

Section List

Attr	Size	Name	Outputs section type, section size, and section name.
DATA	00000010(00000AH)	WORK	
CODE	00000122(00007AH)	PROGRAM	
ROMDATA	00000022(000016H)	CONSTANT	
ROMDATA	00000036(000024H)	F_VECT	

Assemble Error Tag File

Figure A.3 shows an example of an assembler error tag file.

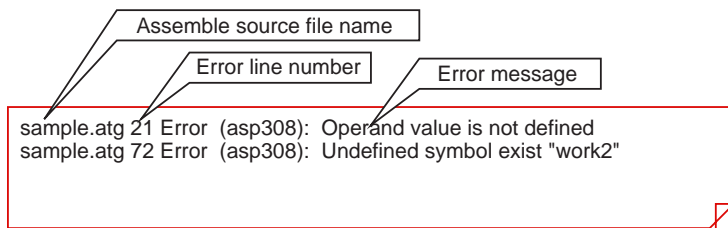


Figure A.3 Example of assembler error tag file

Appendix A-2 Linking(In308)

The following explains the files generated by the linkage editor In308 and how to start up the linkage editor.

Files Generated by In308**(1) Absolute module file (***.X30) ... Generated as necessary**

This file is based on IEEE-695. It consists of the relocatable module files output by as308 that have been edited into a single file.

(2) Map file (*.MAP) ... Generated when option '-M' or '-MS' is specified**

This file contains link information, section's last located address information, and symbol information. Symbol information is output to this map file only when an option '-MS' is specified.

(3) Link error tag file (*.TAG) ... Generated when option '-T' is specified**

This file contains error messages for errors that have occurred when linking the relocatable module files. This file is not generated when no error was encountered. This file allows errors to be corrected easily when it is used an editor that has the tag jump function.

Method for Starting Up In308**>In308 relocatable file name [relocatable file name...] [option]**

Be sure to write at least one file name. The extension (.R30) can be omitted.

Table A.2 Command Options of In308

Command option	Function
-.	Inhibits link processing messages from being output.
-E address value	Sets start address of absolute module file. Always be sure to insert space between option symbol and address value and use label name or hexadecimal number to write address value.
-G	Outputs source debug information to absolute module file.
-L library file	Specifies library file to be referenced when linking.
-LD path name	Specifies directory of library file.
-LOC	Allocates the data of a specified section from a specified address.
-M	Generates map file. This file is named after absolute module file by changing its extension to ".map".
-MS	Generates map file that includes symbol information.
-MSL	The fullname of symbol more than 16 characters are output to map file.
-NOSTOP	Outputs all encountered link errors to the screen. if this operation is not specified, outputs up to 20 errors to the screen.
-O absolute file name	Specifies absolute module file name. File extension can be omitted. If omitted, extension ".x30" is assumed.
-ORDER	Specifies section arrangement and sequence in which order they are located. If start address is not specified, sections are located beginning with address 0.
-T	Outputs error tag file.
-V	Displays version on screen. Linker is terminated without performing anything else.
@Command file name	Starts up In30 using specified file as command parameter. Do not insert space between @ and command file name. This option cannot be used with any other option simultaneously.

Example for Using In308 Commands

Example:
 >In308 SAMPLE1 SAMPLE2 -O ABSSMP
 This command generates ABSSMP.X30.

Extension ".R30" can be omitted.

Command option can be written in uppercase or lowercase as desired.

>In308 @cmdfile
 This command starts up In30 using the content of cmdfile as a command parameter.

#Typical description of cmdfile
 SAMPLE1 SAMPLE2
 SAMPLE3
 -ORDER RAM=400
 -ORDER PROG=0FE0000,SUB,DATA
 -M

Use hexadecimal number to write address. If address begins with alphabet, add '0' at the beginning. Do not add 'H' to denote hexadecimal.

#Relocatable file name
 #Relocatable file name
 #Specifies 400H for start address of RAM section.
 #Specifies sequence in which order sections are located.
 #Command option to generate map file

Add '#' at the beginning of a comment.

Section names are discriminated between uppercase and lowercase.

Link Error Tag File

Figure A.4 shows an example of a link error tag file.

Assemble source file name
 Error / warning line number
 Error message

sample1.a30 87 Warning (In308): sample1.r30: Section type mismatch 'PRO'
 sample1.a30 92 Warning (In308): sample1.r30: Absolute-section is written after the absolute-section 'PRO'
 sample1.a30 92 ERROR (In308): sample1.r30: Address is overlapped in 'CODE' section 'PRO'

Figure A.4 Example of link error tag file

Note: Absolute module files are output in the format based on IEEE-695. Since this format is binary, the files cannot be output to the screen or printer; nor can they be edited.

Appendix

Command input form and command parameters in AS308 system

Map File

Figure A.5 shows an example of a map file.

```
#####
# (1) LINK INFORMATION #
#####
C:\MTOOL\BIN\LN308.EXE -MS SMP
# LINK FILE INFORMATION
SMP (SMP.r30)
Mar 24 16:04:39 1999

#####
# (2) SECTION INFORMATION #
#####
# SECTION      ATR   TYPE      START      LENGTH      ALIGN  MODULENAME
WORK           ABS   DATA      000400      00000A      SMP
PROGRAM        ABS   CODE       FE0000      00007A      SMP
CONSTANT       ABS   ROMDATA    FFE000      000016      SMP
F_VECT         ABS   ROMDAT     FFFFDC      000024      SMP
# Total-----
      DATA      000000A(00000010)Byte(s)
      ROMDATA    000003A(00000058)Byte(s)
      CODE       000007A(00000122)Byte(s)
#####
# (3) GLOBAL LABEL INFORMATION #
#####
WORK           000000

#####
# (4) GLOBAL EQU SYMBOL INFORMATION #
#####
sym2           000000

#####
# (5) GLOBAL EQU BIT-SYMBOL INFORMATION #
#####
sym1           1 000001

#####
# (6) LOCAL LABEL INFORMATION #
#####
@ SMP ( SMP.r30 )
DATA_TABLE     ffe000      DATA_TABLE_END     ffe016      MAIN      fe0066
START          fe0000      WORKRAM_END       00040a      WORKRAM_TOP  000400
addr           000403      char              000400      dummy       fe0079
long           000406      short            000401

#####
# (7) LOCAL EQU SYMBOL INFORMATION #
#####
@ SMP ( SMP.r30 )
AD0  00000380      AD1  00000382      AD2  00000384      AD3  00000386      AD4  00000388      AD5  0000038a
TA2IC 0000006e      TA2MR 00000358      TA3  0000034c      TA3IC 0000008e      TA3MR 00000359      TA4  0000034e
U0MR  00000360      U0RB  00000366      U0TB  00000362

#####
# (8) LOCAL EQU BIT-SYMBOL INFORMATION #
#####
@ SMP (SMP.r30 )
addr_b2      2 000403      char_b0      0 000400      long_b3      3 000406
short_b1     1 000401
```

Link information

Section information

Global label information
This information is output only when command option '- MS' is specified.

Global symbol information
This information is output only when command option '- MS' is specified.

Global bit symbol information
This information is output only when command option '- MS' is specified.

Local label information
This information is output only when command option '- MS' is specified.

Local symbol information
This information is output only when command option '- MS' is specified.

Local bit symbol information
This information is output only when command option '- MS' is specified.

Figure A.5 Example of map file

Appendix A-3 Generating Machine Language File (Imc308)

The following explains the files generated by the load module converter Imc308 and how to start up the converter.

Files Generated by Imc308**(1) Motorola S format file (***.MOT) ... Generated normally**

This is a machine language file normally generated by the converter.

(2) Intel HEX format file (*.HEX) ... Generated when option '-H' is specified**

This is a machine language file generated by the converter when an option '-H' is specified.

Method for Starting Up Imc308

>Imc308 [option] absolute module file name

Table A.3 Command Options of Imc308

Command option	Function
-.	Inhibits all messages but error messages and warning messages from being output to the file.
-E start address	Sets program's start address and generates machine language file in Motorola S format. This option cannot be specified simultaneously with option '-H'.
-H	Generates machine language file in extended Intel HEX format. This option cannot be specified simultaneously with option '-E'.
-ID	Sets ID code of ID code check function . An ID file(extension .id) is created to display ID codes set with this option.
-L	Sets data length that can be handled in S2 records to 32 bytes. Sets Intel HEX format's data length to 32 bytes.
-O	Specifies file name of machine language file generated by Imc30. This file is generated in current directory. Always be sure to insert space between option and machine language file name. Extension of machine language file can be omitted. (Motorola S format .mot; Intel HEX format .hex)
-V	Displays version of Imc30 on screen. Converter is terminated without performing anything else.
-PROTECT1	Sets level 1 of ROM code protect function .
-PROTECT2	Sets level 2 of ROM code protect function .

Example for Using Imc30 Commands

Options are not discriminated between uppercase and lowercase.

Write the option before specifying the absolute module file.

Example
>Imc308 -E 0fe0000 -. DEBUG
This command generates a machine language file "DEBUG.MOT" from the absolute module file "DEBUG.X30" using 0fe0000 as the start address.

Extension ".X30" can be omitted.

>Imc308 -O TEST DEBUG
This command generates machine language file "TEST.MOT" from the absolute module file "DEBUG.X30".

MITSUBISHI SINGLE-CHIP MICROCOMPUTERS
M16C/80 Series
Programming manual <Assembler language>

September First Edition 1999
Edited by
Committee of editing of Mitsubishi Semiconductor
Published by
Mitsubishi Electric Corp., Kitaitami Works

This book, or parts thereof, may not be reproduced in any form without
permission of Mitsubishi Electric Corporation.
©1999 MITSUBISHI ELECTRIC CORPORATION