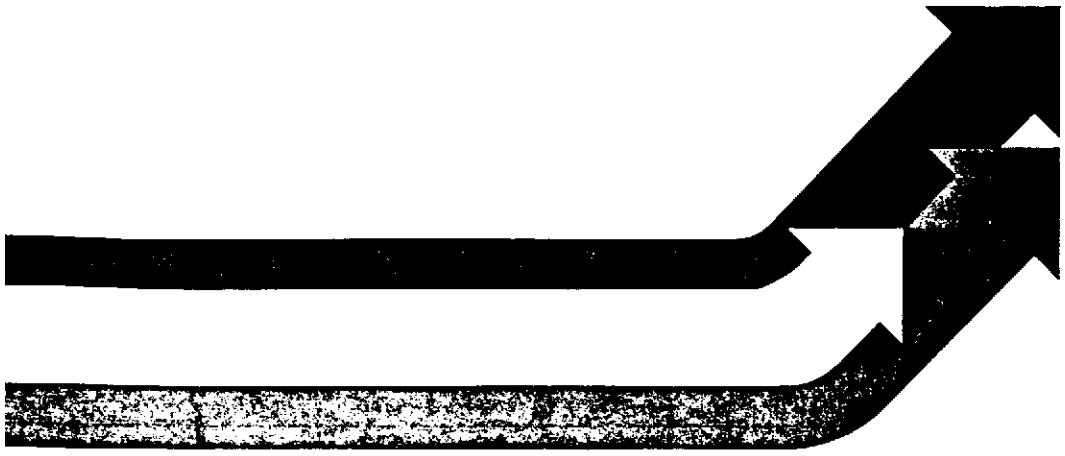
OKI

INSTRUCTION MANUAL

MSM66201

CMOS SINGLE CHIP 8/16 BIT
MICROCONTROLLER



FIRST EDITION
ISSUE DATE : SEP 1991

PREFACE

This manual provides an overview of the MSM66201 and explains its instruction set for programming.

The MSM66201 contains on-chip I/O ports, A/D converters, serial ports, timers, PWM, and other powerful hardware. For information on the functions and control of this hardware, please refer to the *MSM66201 User's Manual*. The explanations presented in this manual assume that the reader understands the contents of the *MSM66201 User's Manual*.

Beyond remembering the hardware configuration of the microcontroller, the first problem encountered by a designer trying to create a program will be how to access the on-chip hardware and memory, and which instructions to use.

The MSM66201 places internal hardware and its control registers in a mapped data memory space. Therefore, control of the MSM66201's on-chip hardware can be considered equivalent to determining how to access its memory space. The fastest way to understand the MSM66201 instruction set is to understand how its memory space is addressed.

Chapter 1, "Overview," explains use and design considerations for the MSM66201's memory configuration and for addressing the hardware mapped in its memory space.

Chapter 2, "Addressing Modes," explains memory space is addressing and its syntax, and provides examples.

Chapter 3, "Details of Instructions," provides a chart of the MSM66201 instructions broken down by function, and it gives detailed explanation of each instruction in alphabetical order. A list of byte counts and cycle counts is also provided.

Chapter 1. Overview

Chapter 2. Addressing Modes

1. Key Features of the MSM66201 Microcontroller

The MSM66201 is a single chip microcontroller (SCMC) specifically designed for efficient programming and high speed data processing.

SCMCs were originally developed to replace control devices for home equipment, particularly electrical and electronic appliances. The internal memory capacity of early SCMCs ranged from 2 to 4K bytes for ROM and from 128 to 256 bytes for RAM. Internal hardware was limited to ports, timers, counters and LCD drivers. The instruction sets available with these early SCMCs included some novel bit manipulations for control; however, the number of arithmetic instructions and addressing modes as well as the quality of these SCMCs were all inferior to what could be obtained with microprocessors.

With the recent development of DRAM and general memory based on large-scale IC technology, the internal memory capacity of SCMCs has increased tremendously, and the CPU processing speed of SCMCs has jumped from several microseconds to several hundred nanoseconds. Consequently the applications for SCMCs are expanding and gradually invading many areas previously occupied by microprocessors. SCMCs are now used for many devices which require high speed processing of large volume of data such as telephone switchboards, high-performance printers, and automobile engine control units.

Such applications for SCMCs have in turn created new functional requirements beyond the need for increased internal memory capacity. These functional requirements must be satisfied by the architecture of future SCMCs, and they are listed below.

1. Increased parallel processing capability
2. Added internal hardware to perform such functions as A/D and D/A conversions, improved communication control, and various display drivers
3. In addition to the hardware development requirements described in item 2, internal hardware readily applicable to the end products is needed
4. Compatibility with large data memory space
5. High speed program processing/low consumption of electrical power

The MSM66201 satisfies these requirements with the following architecture.

(1) Based on Concepts of Application Specific ICs (ASICs)

Future SCMCs must be ASIC oriented as stated in item 3 above. To meet this requirement, the CPU core of the MSM66201 is designed to be independent of I/O (internal peripheral units). Oki Electric Industry Co. Ltd. intends to develop various I/O units which are readily integrated in the end products with the standard CPUs. The great advantages of developing such a family of I/O units are the flexibility of architecture permitted for the product design, and the reuse of the software developed for the standard CPU in all products. The MSM66201 is the vanguard of SCMCs which are developed based on such a concept.

(2) 64K Byte Program Memory Space

The MSM66201 provides a program memory space of 64K bytes. Any address within the memory space is accessible directly or indirectly by CALL or JUMP instructions. The memory space can also store the program constants.

(3) Hierarchy of 64M Byte Data Memory Space

The MSM66201 provides a data memory space of 64M bytes . The memory space consists of three levels, namely banks, pages, and local registers.

The memory space is divided into 256 pages, each holding 256 bytes. The MSM66201 addressing system is designed to place all of the data required for a single processing action on the same page to achieve quick and efficient addressing. To make the data processing within a page effective, each page is divided in 32 areas of 8 bytes, and local registers may be placed in any area.

(4) Complete Bit Manipulation Instruction

The MSM66201 permits individual manipulation of any bit in the data memory space. Also, it is possible to manipulate directly and indirectly any bit in a byte. Transfer between the carry flag(C) and a bit is also possible.

(5) Ease of Array and Pointer Manipulation

The MSM66201 provides 2 index registers of 16 bits and 8 sets of pointing register of 16 bits. These pointers make it possible to access any address of the data memory space. The contents of the registers are automatically saved when a program interrupt occurs and the context switch-over takes effect. Creating data arrays and manipulating pointers is easy on the MSM66201.

(6) Organized Instructions

In the MSM66201 design, the internal hardware and registers are allocated to the memory space, and numerous addressing modes to access data are available; therefore, it is possible to manage and control all resources of the MSM66201 with a few instructions. The transfer and arithmetic instructions are plain and clear, consequently program coding is easy, and programs produced are easy to understand.

(7) Extremely Quick Switching of Program Context

The MSM66201 design permits extremely quick switching of the program context when a program interrupt handler functions or a task switching is executed in a multitasking program.

(8) Memory Mapped I/O

In the MSM66201, the internal I/O units such as I/O ports and timers are allocated to the data memory space (memory mapped I/O). Consequently no specific input and output instructions are required; these instructions are replaced by the data addressing instructions. This scheme produces not only a simple instruction system but also a flexible design that makes adding and modifying of internal I/O hardware easy. The MSM66201 is really developed to meet the ASIC requirements of the future, and the memory mapped I/O design enhances the ASIC features of the MSM66201.

2. Memory Configuration

The memory space of the MSM66201 is divided into the program and data memory spaces which are described in subsections 2.1 and 2.2 respectively. In subsection 2.3, the concept of data segments that are allocated to the memory space for assembler operation is described.

[NOTE] The program memory space (or program space) is sometimes called ROM (Read Only Memory) space because ROM is usually its main constituent element. Likewise the data memory space (or data space) is sometimes called RAM (Random Access Memory) space because RAM is usually its main constituent element.

2.1 Program Memory Space

2.1.1 Structure of the Program Memory Space

The program space of the MSM66201 accommodates 64K bytes (0000_H to $FFFF_H$) maximum; 40 bytes (0000_H to 0027_H) of which forms the vector table space composed of 20x2-byte segments. Another 16 bytes (0028_H to 0037_H) form the VCAL table space of 8x2-byte segments as shown in Figure 1-1.

(1) Vector Table Space

The vector table space saves the vector addresses when the reset instructions (reset input, BRK and WDT) are executed or interrupts are caused by the peripheral hardware. Each vector address is saved in 2 bytes (16 bits); the lower and upper bytes of the vector address are stored respectively in the lower and upper addresses of the RAM. The vector table is described by the assembler with pseudo instructions. (See the following examples)

Examples of Assembler Coding for Vector Table

```
        CSEG  AT  0000H
;
        DW    START          ----------  0000H
        DW    BRK_RESET      ----------  0002H
        DW    WDT_RESET      ----------  0004H
        DW    NMI_ENTRY      ----------  0006H
        DW    INT0_ENTRY     ----------  0008H
                :
        DW    INT1_ENTRY     ----------  0026H
```

Please note that at a reset or interrupt, the contents of the concerned vector table (vector addresses) are set in the PC; no jump instruction into the vector table space is specified. Also, it is mandatory to allocate addresses 0 through 1 to the vector addresses for reset input, but the remaining addresses (2 through 27_H) need not be specified when no corresponding reset and interrupt instructions are executed, and the unused area may be used to enter programs.

The allocation of the vector table is shown in Table 1-1.

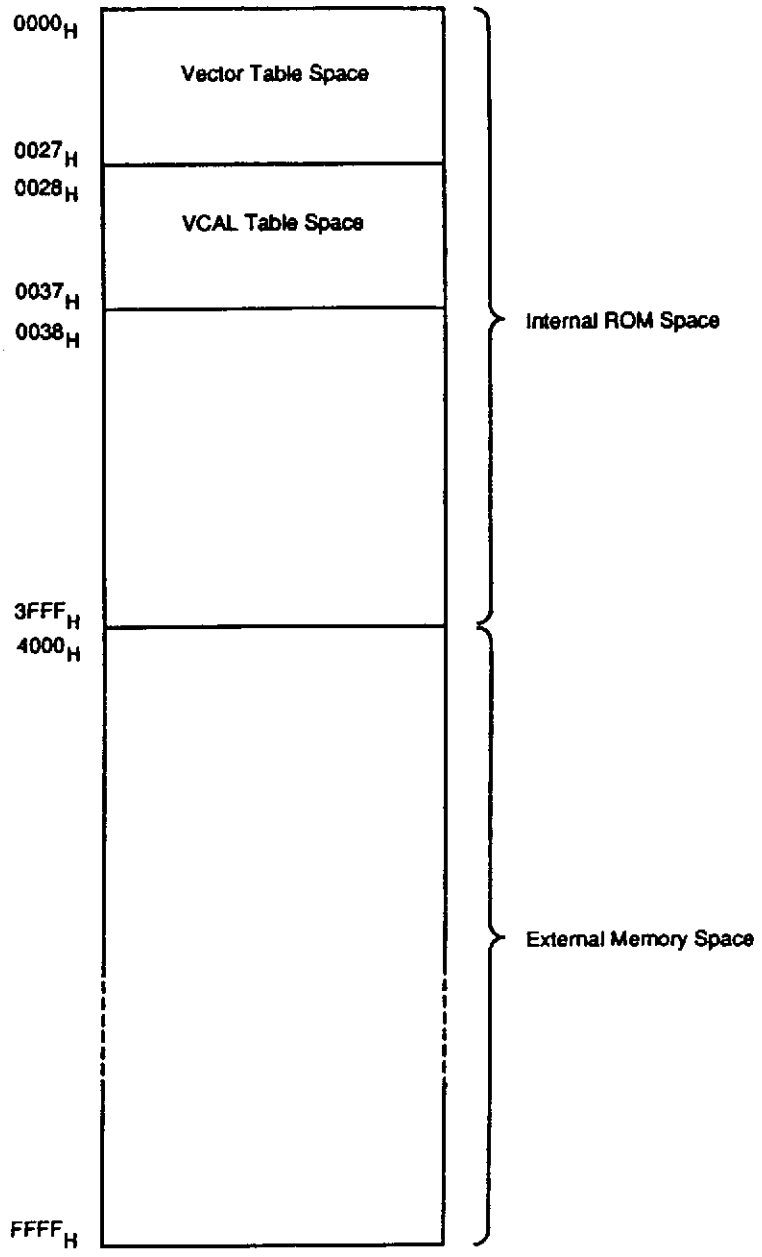


Figure 1-1. Structure of Program Memory Space

Table 1-1. Allocation of Vector Table

Address	Corresponding reset process or interrupt
0000_H	Reset by RESET input
0002_H	Reset by BRK instruction
0004_H	Reset by WDT / Reset by operation code trap function
0006_H	Interrupt by NMI pin input
0008_H	Interrupt by INT0 pin input
$000A_H$	Interrupt by serial port readying for receiving
$000C_H$	Interrupt by serial port readying for transmit
$000E_H$	Interrupt by BRG for serial port receive
0010_H	Interrupt by timer 0 overflow
0012_H	Interrupt by event occurrence of timer 0
0014_H	Interrupt by timer 1 overflow
0016_H	Interrupt by event occurrence of timer 1
0018_H	Interrupt by timer 2 overflow
$001A_H$	Interrupt by event occurrence of timer 2
$001C_H$	Interrupt by timer 3 overflow
$001E_H$	Interrupt by event occurrence of timer 3
0020_H	Interrupt by end of A/D conversion
0022_H	Interrupt by PWM timer overflow
0024_H	Interrupt by BRG for serial port transmit
0026_H	Interrupt by INT1 pin input

[NOTES] 1. The preceding allocation is applicable only for the MSM66201. The MSM66201 is designed as the basic unit of a family of microcontrollers (nX8/300 series) that is based on ASIC concepts, and the MSM66201 is marketed as a standard product. Therefore it is possible to install customer specific internal I/O units which are different from the standard MSM66201 as long as the CPU remains the same. For such cases, the allocation of the vector table may change. Also the number of interrupts may increase beyond the standard memory space (0000_H to 0027_H) capacity; the excess interrupt vector addresses must be allocated behind the VCAL table space(following 0038_H).

2. When an address of 16-bit length is stored in data memory, in a register, or is transferred, the lower byte of the data is stored in the lower address of the memory or register and the upper byte is stored in the upper address. This rule is implicitly applicable in the MSM66201 device unless it is stated otherwise.

(2) VCAL Table Space

The MSM66201 uses a vector call instruction (VCAL) which is one byte long. The addresses of the VCAL table (0028_H through 0037_H) are entered in the operand of the VCAL instruction. When the VCAL instruction is executed, the vector address of the VCAL table specified by the operand is set in the PC after the return address is saved in the system stack.

The VCAL table is entered by a DW pseudo instruction like the vector table. When VCAL instructions are not used, the VCAL table area can be used for entering programs.

(3) Conventional Coding of Program Space

As stated in the articles (1) and (2), the address area from 0000_H to 0037_H is allocated to the tables. Also, as stated in (1), the area excluding from 0000_H to 0001_H may be used for entering

programs, however it is recommended that this be avoided. Enter programs in the addresses higher than 0038_H, because later program changes may require the addition of instructions for interrupts and VCAL.

The following three examples are produced using these rules.

Conventional Coding of Program Space

```
MAIN    SEGMENT  CODE
CSEG    AT   0000H
;
;       ******************
;       *  Vector Table  *
;       ******************

DW      START           ; Reset
DW      BRK_RESET       ; Break instruction

        :

DW      INT1_ENTRY      ; INT1 Interrupt
```

Coding Vector Table

```
;       ******************
;       *  VCAL Table    *
;       ******************

VCAL0:  DW          ERROR_PROCESS
VCAL1:  DW          PORT_INIT_PROCESS
VCAL2:  DW          RAM_INIT_PROCESS

DS      10              ; Reserved for future
```

Coding VCAL Table

```
;       ******************
;       *  Main Routine  *
;       ******************

RSEG    MAIN
;
START:
        :
        :
        :
        :
END
```

Program Body

2.1.2 Accessing Program Memory Space

Accessing program memory space is performed in two ways: by PC, or by instruction.

(1) Access by PC

Accessing the program memory by using the PC is usually performed when a program is being executed. The PC always contains the address of the memory where the next instruction to be executed is stored. The contents of the address indicated by the PC is the instruction. Following the execution of an instruction, the contents of PC is automatically increased an amount equal to the number of bytes of the executed instruction by hardware. When a reset input is given from an external pin or an interrupt occurs, the PC contents is replaced by the value stored in the vector table. These operations are done by the hardware; the programmer need not be concerned with them. However, the PC count can be changed by the programmer if necessary. For instance, it is possible to replace the PC contents with the value stored in the VCAL table by executing a VCAL instruction. Also, the PC contents can be replaced by the addresses specified by the operands of the following instructions upon their executions.

Instructions that Can Replace PC Contents

SJ, J, JC, JBS, JBR, JRNZ, SCAL, CAL

Refer to Chapter 3 for details of these instructions.

(2) Access by Instruction

The MSM66201 provides instructions that allow accessing the program memory space.

Instructions to Access Program Space

LS, LCB, CMPC, CMPCB

With the use of these instructions, it is possible to transfer the contents of the program memory to the accumulator or compare the program memory to the accumulator.

The following addressing modes are available for specifying the locations of memory space when these instructions are used.

- Direct Addressing: specify the address in the space of 64K bytes directly using immediate addressing.
- Simple Indirect Addressing: specify the address indirectly according to the contents of data memory such as a local register (er0 to er3), pointing register (DP, X1, X2, USP), or system stack pointer (SSP).
- Double Indirect Addressing: specify the address indirectly according to the content of the data memory which is addressed indirectly by the contents of a pointing register (DP, X1, X2, USP).
- Indirect addressing with a 16-bit base: specify the address by offsetting from the base address in the 64K byte memory space; the amount of offset is specified by the contents of a pointing register (DP, X1, X2, USP) or the contents of another data memory.

For the details of the instructions and addressing modes, refer to Chapter 3 and Section 2 of Chapter 2 respectively.

2.2 Data Memory Space

2.2.1 Structure of Data Memory

(1) Concept of Page

This article explains the concept of *pages*.

Generally, the data memory space used for a program module or unit process is not so large. In other words, the upper bits of the addresses of the data in the space which are addressed during a unit process hardly change. So, if the data memory space is divided into blocks of a appropriate size for a unit process, and all of the data involved in a program is stored in one block, addressing the data for the process may be achieved by simply offsetting each data from the base address of the block. Such a configuration of data memory space will shorten the access time as well as the machine codes.

Using such concepts, the architecture of the MSM66201 divides each data memory bank into blocks of 256 bytes. This block is called a *page*. Thus, a bank of 64K bytes consists of 256 pages of 256 bytes.

In Figure 1-2, the configuration of the data memory is shown.

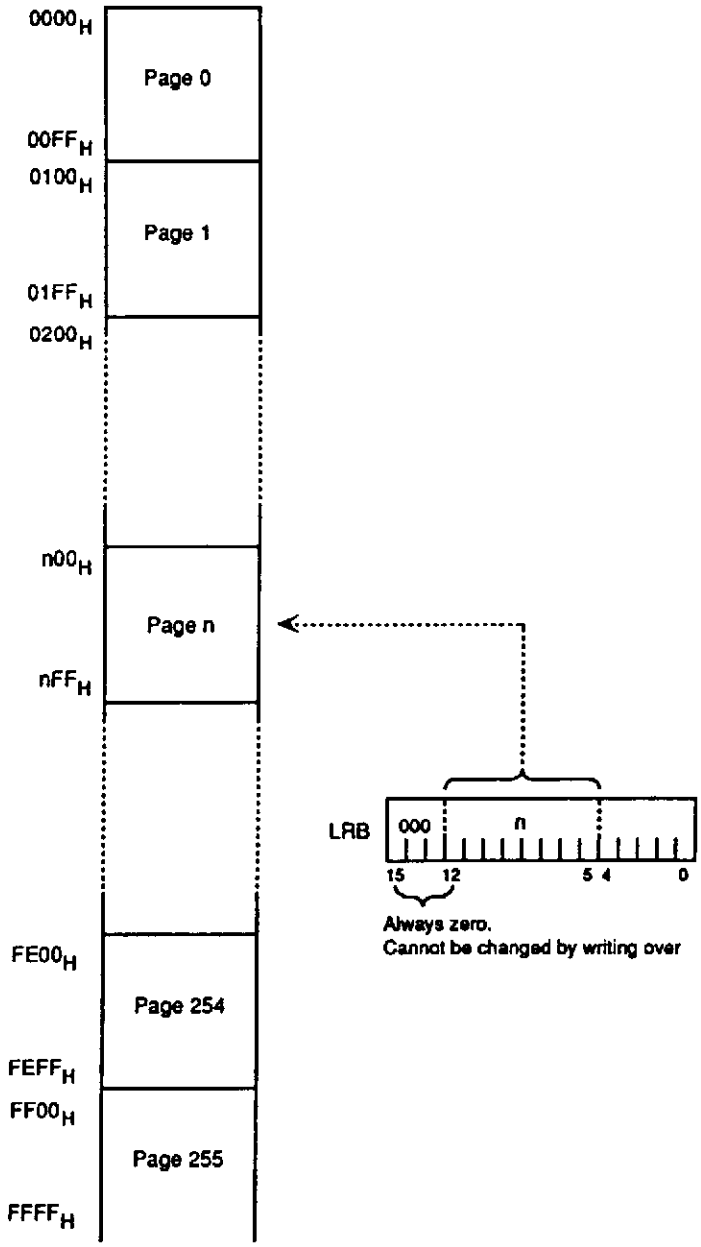


Figure 1-2. Page Configuration of Data Memory

As shown in Figure 1-2, the pages are numbered progressively, starting at the lowest address, as Page 0, Page 1, Page 2, up to Page 255. The page numbers used are specified by bits 12 through 5 of LRB, and the page thus specified is called the current page.

When only the on-chip data space of the MSM66201 is used, the corresponding addresses used are 0000_H through $027F_H$, and the only corresponding pages used are Pages 0, 1, and 2.

Furthermore, only 128 bytes are useable in Page 2. The configuration of the data memory space used for on-chip space only is shown in Figure 1-3.

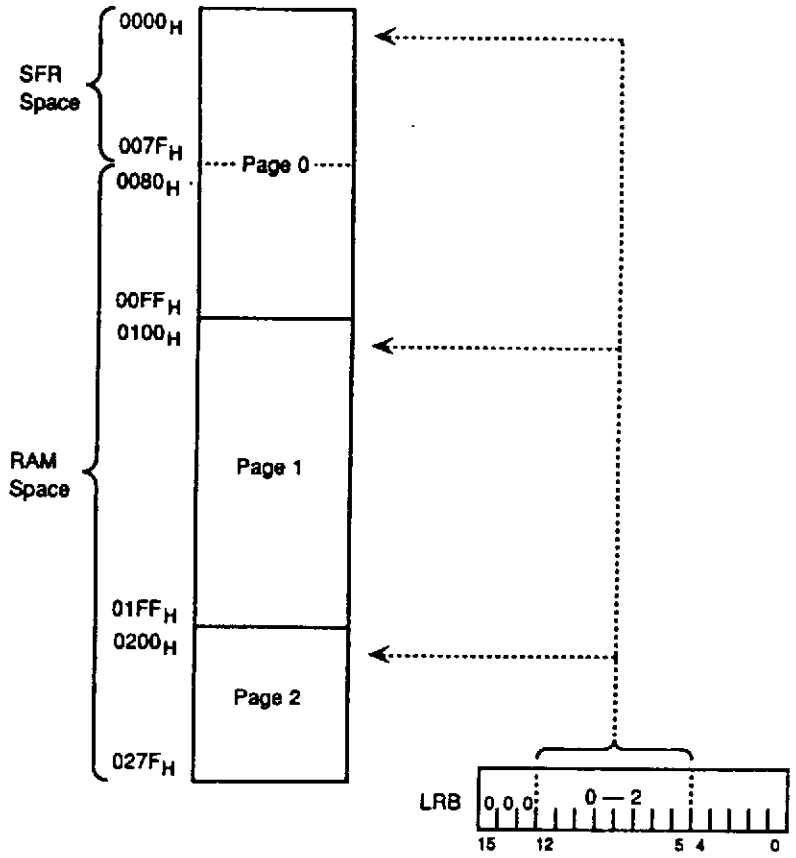


Figure 1-3. Page Configuration of Data Memory (On-Chip Space)

Addressing by offsetting within the current page is called current page addressing; its details are given in Chapter 2. Current page addressing is allowed with most instructions in the MSM66201 instruction set, and our chip designers made their best effort to realize very fast memory access with current page addressing. This fits the concept that the data space for each module should be located within one page, and in that situation current page addressing would be used most frequently.

(2) Configuration of Page 0

All of the pages (256 total) thus defined are basically similar except for Page 0. Initially, the special function register (SFR) and the pointing register (PR) are allocated to Page 0. (See Figure 1-4)

Secondly, to access the contents of Page 0, zero page addressing, which does not depend on LRB, is available. This is in addition to the familiar current page addressing, which is LRB dependent.

These items are further explained in the following paragraphs.

SFR

The internal hardware of the MSM66201 will be accessed at every stage of a program, therefore it is desirable to allocate the hardware to page 0 in a memory-mapped I/O configuration to increase the program efficiency. The configuration of memory-mapped I/O requires fewer instructions to control the hardware units compared to the I/O-mapped I/O configuration, and it also provides the flexibility to modify the internal hardware while the CPU stays unaltered (one of the beneficial features of ASICs).

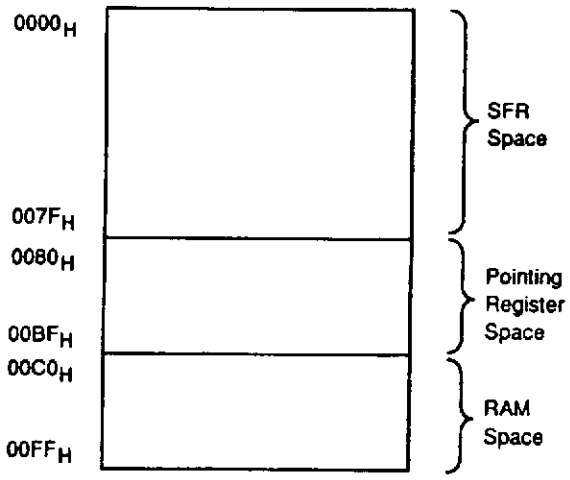


Figure 1-4. Configuration of Page 0

Considering the foregoing factors, the internal hardware of the MSM66201 such as timers, counters, A/D converters, and so on, and the registers which control the memory-mapped I/O hardware, are allocated to the addresses from 0000_H to $007F_H$ of Page 0. The general term for these hardware units is "Special Function Register" or SFR for short; the latter abbreviation is used in this book.

SFR includes, in addition to timers, A/D converters and so on, the general purpose registers which are listed in the following table; these are used for arithmetic and memory management.

General Purpose Registers Included in SFR

SSP	System Stack Pointer
LRB	Local Register Base
PSW	Program Status Word
ACC	Accumulator

Pointing Register (PR)

The pointing registers are allocated to addresses 0080_H through $00BF_H$ of Page 0. They are 16-bit registers and function as pointers when addressing the program or data memory.

The pointing register is composed of the following four registers which form a set.

X1 ..... Index Register 1
X2 ..... Index Register 2
DP ..... Data Pointer
USP ..... User Stack Pointer

Eight pointing registers or eight sets of the foregoing units are allocated to Page 0 and they are named PR0, PR1, PR2 ... PR7. (See Figure 1-5).

For the details of pointing register's function, refer to Section 3 of this chapter and the description of addressing modes in Chapter 3.

As shown in Figure 1-5, selecting the register out of eight units is performed by the lower 3 bits of PSW whose group name is System Control Base (SCB).

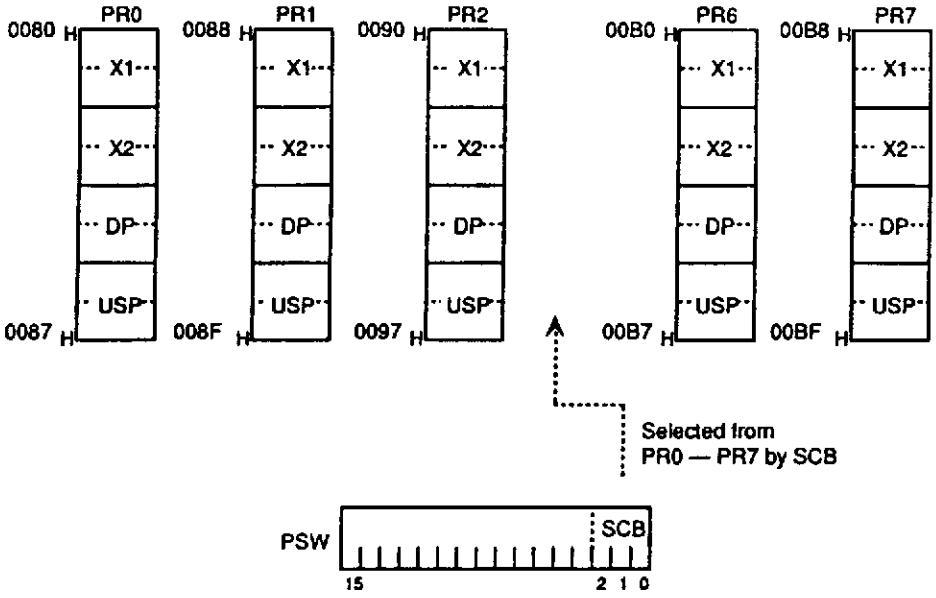


Figure 1-5. Pointing Registers

Zero Page Addressing

As described in (1), the data memory space of the MSM66201 is divided into pages, each of which contains 256 bytes, and the work area of a program module is basically contained within a page. SFR and the pointing registers are contained within Page 0; however, these units are expected to be referred to throughout various stages of program execution, resulting in a changing value of LRB. Such a situation degrades program efficiency. To avoid this, zero page addressing which does not depend on LRB is provided for the data within Page 0. This addressing mode is in addition to universal current page

addressing. The practical application of zero page addressing is described in the *Addressing Modes* in Chapter 2.

As shown in Figure 1-6, addresses 0000_H through $00FF_H$ of Page 0 are RAM. When these addresses are used for space to store global variables like the external variables of a C compiler, zero page addressing can be employed to enhance program efficiency.

(3) Local Registers

As described in the articles (1) through (2), the data memory space is stratified in 256 pages. The upper eleven bits, from 12 to 5, of LRB are used to identify the banks and pages, and the remaining lower five bits of the 13-bit register are used to set the base for local registers.

As described in (1), the data memory space is divided into 256-byte pages. Each page provides register space in 8-byte units. The architecture of the MSM66201 permits the allocation of registers to any convenient addresses within the current page rather than specific fixed addresses. Such registers are called *local registers* and their base addresses are specified by the lowest five bits of LRB. (See Figure 1-6)

The local registers are used to store the data when transfer or arithmetic instructions are executed. The unit of the register can be either byte or word. For byte unit operations, each register forms 8 bits x 8 register sets, and the data is entered starting at the lowest, as r0, r1, r2 . . . r7. The corresponding configuration for word operations is 16 bits x 4 register sets with data entry starting at the lowest, as er0, er1, er2, er3.

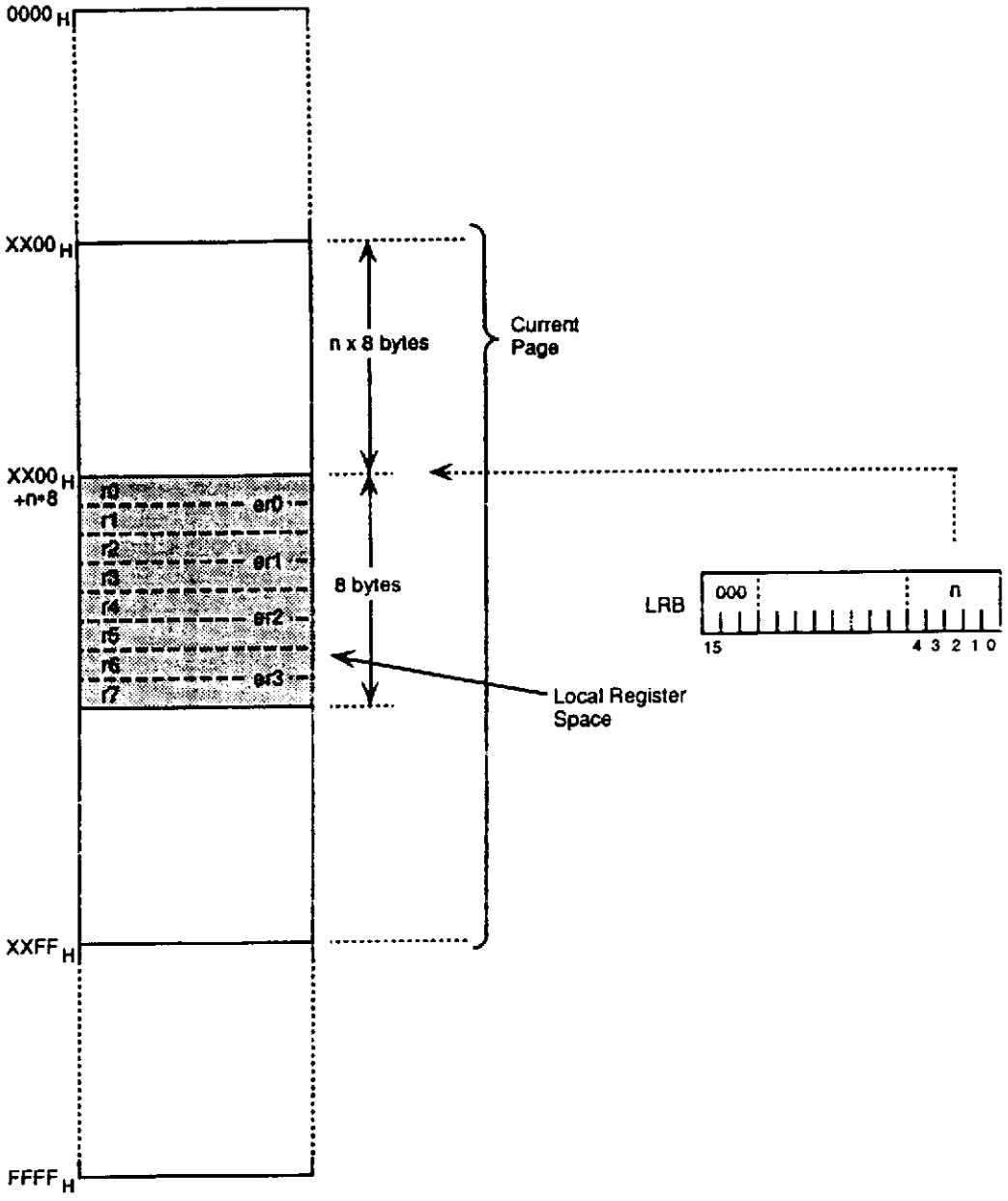


Figure 1-6. Local Register Space

2.2.2 Accessing Data Memory Space

There are three kinds of data memory accessing:

(1) Addressing the general data space

(2) Addressing the system stack space

Detailed descriptions follow.

(1) Accessing the General Data Space

The general data space is all data space other than the spaces which are allocated to the system and user stacks. (Note that the general space includes SFRs, which include ACC, PSW, LRB, etc.)

All addressing of the general data space is performed by the instructions. More than half of the instructions used by the MSM66201 are for addressing the general data space. They are listed below. The details of these instructions are described in Chapter 3, *Details of Instructions*.

Instructions for Addressing the General Data Space

• Data transfer instructions	(Load)	L, LB	(Store)	ST, STB
	(Move)	MOV, MOVB	(Clear)	CLR, CLRB
	(Exchange)	XCHG, XCHGB, XNBL		
• Rotate/Shift instructions	(Rotate)	ROL, ROLB, ROR, RORB		
	(Shift)	SLL, SLLB, SRL, SRLB, SRA, SRAB		
• Increment/Decrement instructions		INC, INCB, DEC, DECB		
• Arithmetic operation instructions	(Multiply)	MUL, MULB	(Divide)	DIV, DIVB
	(Add)	ADD, ADDB, ADC, ADCB	(Subtract)	SUB, SUBB, SBC, SBCB
• Logic operation instructions	(Logical AND)	AND, ANDB	(Logical OR)	OR, ORB
	(Exclusive OR)	XOR, XORB		
• Compare instructions	CMP, CMPB			
• Bit manipulation instructions	(Set bit)	SB, SBR	(Reset bit)	RB, RBR
	(Test bit)	TBR	(Move bit)	MBR

Almost all of the addressing instructions may operate in word length (16-bit length) or byte length (8-bit length) modes. Each instruction accommodates these operations with two distinctive codes like MOV and MOVB, or ADD and ADDB. In Chapter 3, the details of the instructions are given.

In some instructions primarily for the accumulator, the same machine codes, depending on the flag called *data descriptor* (DD), perform both byte and word operations. This flag is allocated in PSW, and constitute the most notable features of the MSM66201 control system with their ability to provide multiple instructions. The functions of these two flags are detailed in 2.2.4.

This section provides additional information about addressing modes for the general data memory. Although the previously described instructions perform the addressing, the programmer must enter the mnemonic codes (MOV, ADD, CMP, etc.) in the assembler source program. These mnemonic codes define the operations, and the objects of the operations are the addresses of the data space specified by the operands following the mnemonic codes.

For example, transfer instruction MOV is entered as

```
MOV operand_1, operand_2
```

This instruction orders the transfer of the contents of operand_2 into operand_1 using word units. Operand_1 and operand_2 specify the addresses that are the objects of the operation. There are various ways to specify or note the addresses of the objects. Addressing notation defines how to access the data space, or in other words, defines the addressing mode.

Two of the addressing modes for the general data space have been described in 2.2.1. They are listed below, and further detailed in Chapter 2.

1. Current page addressing
2. Zero page addressing

In zero page addressing, the address or symbol which is the object of an operation is entered as it is. In current page addressing, a descriptive word *off* for addressing is placed preceding the address or symbol which is the object. For example, to transfer in word length units the data space which is labelled WORK_1 in the current page to the address $A0_H$ of Page 0, the following code is used.

```
MOV 0A0H, off WORK_1
```

Current page and zero page addressing modes are the basic modes; however, the following additional addressing modes are available to enhance program efficiency as listed below.

3. Register addressing
4. Pointing register indirect addressing

The register addressing is specifically used to access the general purpose registers within Page 0. The registers include SSP, LRB,PSW, ACC, X1, X2, DP and USP and the local registers in the current page (r0, r1, r2, r3, r4, r5, r6, r7 or er0, er1, er2, er3). The name of the specific register is entered in the statement except for ACC which changes to A. For example, the following entry will transfer er0 to ACC.

```
MOV A, er0
```

Pointing register indirect addressing is used for programs that require accessing arrays allocated in the data space with the use of pointers, or for accessing data that can be stored indirectly with pointers to improve overall program efficiency. There are three variations to this addressing mode.

The first variation is indirect addressing with a base using X1 and X2. Array-type data are conveniently handled with this access scheme in which the amount of offset from the base is processed by X1 and X2. For example, the following entry will transfer the 10th array element, SEG_DATA[10], in the array SEG_DATA to the accumulator after X1 is set to 10.

```
MOV A, SEG_DATA[X1]
```

The second variation is indirect addressing by the data pointer (DP) that is used as a pointer for general indirect addressing. For example, the following entry will move the data of a byte length pointed to by DP to local register r0.

```
MOVB r0, [DP]
```

The third variation is indirect addressing by USP. The effective address is generated by USP and the signed (±) displacement.

```
MOVB A, +2[USP]
```

The access modes for the general data memory space have been outlined in the preceding sections; however, the functional details of the instructions and the thorough descriptions of the addressing modes are not given. Refer to Chapter III for functional details of each instruction. Refer to Chapter II for details on addressing.

There is one more addressing mode for the general data space which is termed *bit addressing*. Bit addressing, as well as the bit segment, will be described in 2.3, "Concept of Segments".

(2) Accessing the System Stack Space

The system stack is the stack space that is indirectly accessed by system stack pointer (SSP). Accessing the system stack space is performed in the following cases.

1. SCAL, CAL and VCAL instructions are executed
2. Interrupts occur
3. PUSHS instruction is executed
4. RT and RTI instructions are executed
5. POPS instruction is executed

The system stack space is generally called *stack space*, and it is used to save the return address when call instructions are executed (case 1). It is also used to save the return address and the contents of ACC, LRB and PSW when interrupts occur (case 2). The system stack space can be used to stack PR, ACC and LRB directly using the PUSHS instruction (case 3). The RT instruction returns the return address to PC which is saved in the system stack space by CALL instructions (case 4). RTI instruction is used to return from interrupts and to return PSW, LRB, ACC and the return address from the system stack. The POPS instruction is used to pop the data in the system stack which has been put in by PUSHS instruction (case 5).

2.2.3 Word Boundary

In 2.2.2, three categories of accessing the data space are established:

1. Accessing the general data space
2. Accessing the system stack

In category 1, it has been said that accessing the general data space can be entered in word unit or in byte unit. When the word unit is used, the existence of the word boundary must be observed.

The word boundary is the borderline between word units, and it follows behind every odd-numbered address. As shown in Figure 1-7, there is no word boundary between addresses 0 and 1, hence a word that consists of addresses 0 and 1 is permissible. On the other hand, there is a word boundary behind address 3 because it is an odd-numbered address. Therefore, a word that consists of addresses 3 and 4 crosses the word boundary, and such a word is not usable for the accessing operation. In short, each word must start at an even-numbered address.

Using the transfer instruction, MOV as an example, the operation of the CPU core is explained. When the contents of addresses in the data memory space are transferred to the accumulator by zero page addressing, the following entry will transfer the contents of the addresses $C0_H$ and $C1_H$ to the accumulator.

```
MOV A, 0C0H
```

The corresponding machine code is given as:

B5 C0 99

The C0 in the second byte corresponds to the address of the second operand. When these codes are executed by CPU, the contents of the two bytes which starts at address $C0_H$ will be transferred.

If the $C0_H$ of the example address is replaced with $C1_H$, the entry is changed as shown.

```
MOV A, 0C1H
```

As mentioned previously, the words used in the access operation must start at an even-numbered address and be 2 bytes wide. However the $0C1_H$ of the last entry is an odd-numbered address which creates a logical inconsistency, and consequently the assembler issues a warning. The assembler, however proceeds to change the address code as shown.

B5 C1 99

When the CPU executes these codes, the process is identical to the original one where the two bytes of data in addresses $0C0_H$ and $0C1_H$ are transferred to the accumulator. Thus the combination of $0C1_H$ and $0C2_H$ is ineffective, because the CPU forces the LSB (lowest bit) of the effective address to be 0 in the accessing process.

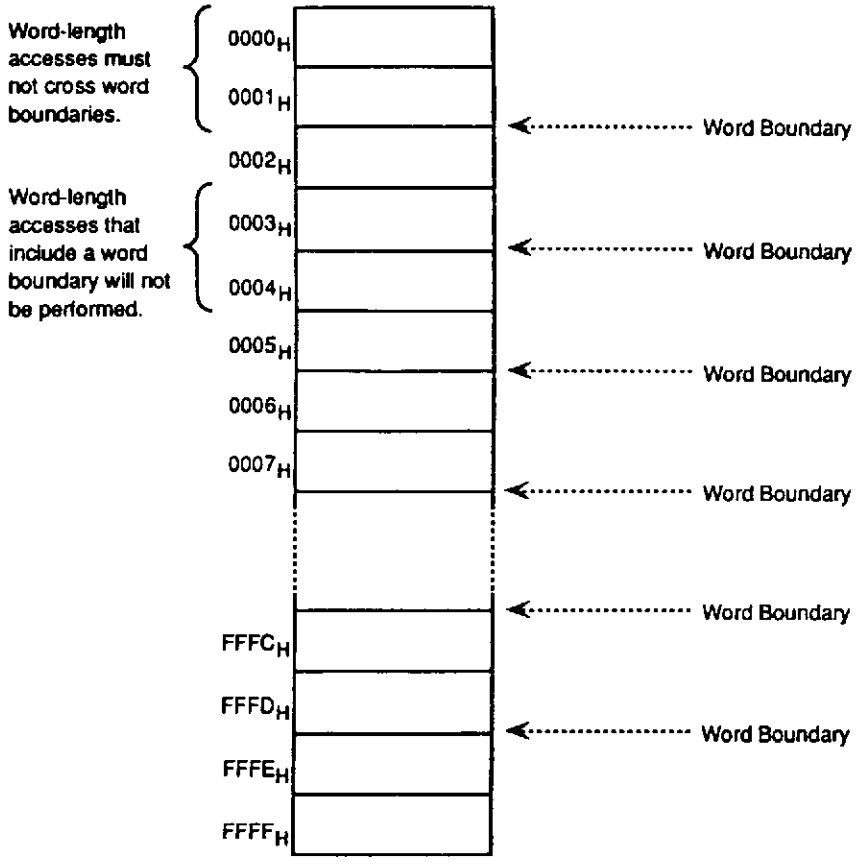


Figure 1-7. Word Boundary

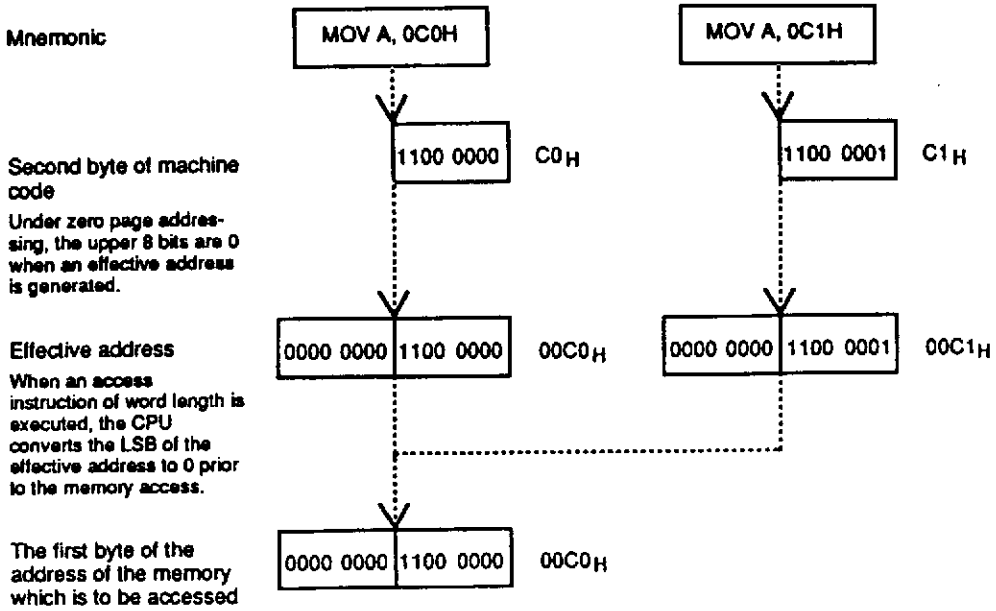


Figure 1-8. Address Generation Process in Accessing Word Long Data Space

Instructions that are Influenced by Word Boundary

The instructions that are influenced by word boundaries are specified here. Word boundaries exists for almost all the instructions that access the general data space in word length units, and for all instructions used to access the system stack space. These instructions are listed in Table 1-3-1 and Table 1-3-2.

Table 1-3-1. Instructions Influenced by Word Boundary

• Data transfer instructions	L, ST, MOV, CLR, XCHG
• Rotate/Shift instructions	ROL, ROR, SLL, SRL, SRA
• Increment/Decrement instructions	INC, DEC
• Arithmetic operation instructions	ADD, ADC, SUB, SBC
• Logic operation instructions	AND, OR, XOR
• Compare instructions	CMP

Table 1-3-2. Instructions Influenced by Word Boundary

• Call instructions	SCAL, CAL, VCAL
• Return instructions	RT, RTI
• Push/Pop for system stack	PUSHS, POPS
• Save into system stack at interrupt	

The instructions unaffected by the word boundary are the instructions for accessing the user stack PUSHU, POPU), and ROM addressing.

Thus, it is possible to push the data in the user stack without regard to their order; the data may be in byte or word length. Also, it is possible to refer to the ROM table starting at an odd-numbered address.

When a programmer codes a source program, the addresses referred to are seldom entered in numbers such as shown below:

```
MOV A, 0C0H
```

The addresses displayed in the operands are usually symbols which are defined in the data segment as labels or defined by DATA pseudo instructions. The data segment is the terminology used for assembler, and its concept is described in detail in 2.3. For the time being, it may be considered to be synonymous to the data memory space.

When the data space of a word unit is entered in a program, the data segments become meaningful. The data segments that contain the work space accessed by word-unit instructions must meet the requirement that the starting bytes (addresses) be even-numbered. In other words, labels for data segments used for the operand of an access instruction that uses word units must be assigned to even-numbered addresses.

When an odd-numbered address is entered for the operand of an access instruction that operates with words in the general data space, the assembler issues an warning message for errors only when the operand is evaluated as an absolute value.

When the operand is evaluated as a relocatable value, the symbol for the operand has the attribute of a relocatable data segment, and the real value of the relocatable value or symbol can not be known until the memory allocation is completed by the linker. This means that the assembler itself is unable to check the value. The currently available linker (RL66K) lacks the capability to perform such a checking function. (Linker improvement is is planned.)

The starting byte of the data segment that is accessed by the instructions which operate on words must be able to instruct the assembler to allocate an even-numbered address. Therefore, the relocatable assembler (RAS66K) interprets the pseudo instruction segment to define segments with the added requirements for word attributes (no crossing of word boundaries). The segment with word attributes is assigned so that the starting address is always an even number.

This checking of word boundaries is performed only for cases in which the addressing entry in the operand indicates directly the accessing object as is done in page addressing. Addressing the general data space can also be done by other methods such as indirect addressing by pointing register, where the operand is entered with an indirect address. For such a case, the assembler is unable to check the word boundary because the assembler needs to know the contents of the register used for indirect addressing prior to checking the word boundary. For the assembler to know the contents of a register it is necessary to check the entire program flow, and this is far beyond the assembler's function. Therefore, managing the word boundary in indirect addressing is entirely the responsibility of the programmer.

2.2.4 Data Descriptor (DD)

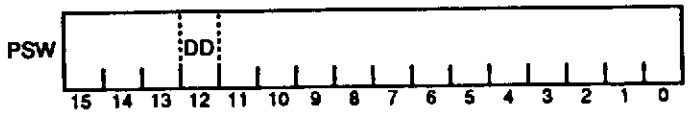
MSM66201 provides a special flag called data descriptor (DD)

DD is associated with manipulation and transfer instructions for the accumulator of the MSM66201. DD affects the length of data accessed.

DD flag is described in the following articles.

(i) DD allocation

DD is allocated to bit 12 of the program status word (PSW).



(ii) Function of DD

Prior to the description of the function of DD, the mnemonic used for the MSM66201 will be explained. The instructions for the MSM66201 are generally classified in two groups: the instructions that operate on almost all of the resources in word length units, and the remaining instructions that operate in byte units. The mnemonic expressions for the latter group of instructions are distinguished from the former with suffix B as shown in the following examples.

Word long MOVE instruction : MOV
Byte long MOVE instruction : MOVB

Word long ADD instruction : ADD
Byte long ADD instruction : ADDB

Word long logical AND : AND
Byte long logical AND : ANDB

The corresponding machine codes for the word based and byte based instructions are generally different. The exception to this is the instructions which operate on the accumulator where the same machine code is used for both instructions of word and byte unit in conjunction with the use of DD. With DD=1, the instructions operate in word length, and with DD=0, they operate in byte length. Such instructions influenced by DD are listed in (iii).

(iii) Instructions influenced by DD

ST / STB	A , obj	store in accumulator instruction
SWAP / SWAPB		swap instruction for accumulator
ROL / ROLB	A	left rotation instruction for accumulator
ROR / RORB	A	right rotation instruction for accumulator
SRA / SRAB	A	right arithmetic shift instruction for accumulator
SLL / SLLB	A	left logical shift instruction for accumulator
SRL / SRLB	A	right logical shift instruction for accumulator

The term *obj* above represents the addressing entry (like r0, er0, [DP], 100H and off 0C0H) which is legally used as the operand of the instruction. Note that all instructions are for operations that use the accumulator.

The machine code for each pair of the instructions listed above is the same. Basically, for an instruction, the mnemonic and the machine code form a one to one correspondence. The assembler of the MSM66201, however, allows two mnemonic terms for a machine code.

For example, the machine code 88 causes the contents of the accumulator to be stored in local register 0. With DD=1, the instruction operates in word units; the contents of the accumulator are extended to 16 bits and transferred to the local register er0. With DD=0, the operation is in byte units, and the lower 8 bits (A_L) are transferred to the local register r0. The mnemonic statements for these instructions follow.

```
ST   A, er0      word length operation

STB  A, r0       byte length operation
```

Providing two mnemonic terms for one machine code has two advantages. The first advantage is that the programmer can transmit his intention to the assembler: "the operation should be in word or byte length." The second advantage is that when the programmer or a third person reviews the source program, the unit of the operation is easy to see.

Even if the assembler understands the programmer's intention, the assembler must have some capability to verify the unit of operation. For instance, the programmer intends to use byte units in the store instruction and enters in the source program as shown. However, if DD=1 at the time of

```
STB  A, r0
```

program execution, the CPU will execute the instruction in word unit contrary to the programmer's intention. It is desirable for the assembler to provide some security means to verify the consistency between the programmer's intention and the DD value. However, the assembler cannot obtain the DD value because DD is specified by the program as explained later in detail.

As a supplementary checking means, the assembler provides the USING DATA pseudo instruction by which the programmer indicates to the assembler the value of DD to be used. To implement this process, the programmer enters

```
"USING DATA WORD"
```

in the source program which is equivalent to the message "Following this statement, DD=1 in this program" by the programmer. Similarly, the entry of

```
"USING DATA BYTE"
```

is equal to the message "Following this statement, DD=0 in this program". When DD is given by the USING DATA instruction, the assembler verifies the DD dependent instructions for consistency between the data unit of the instruction and DD; an error warning is issued if inconsistencies are found.

If no such verification of DD consistency is required, the statement "USING DATA ANY" is entered in the program. The USING DATA statement provides some degree of verification of consistency regarding the data length. However, it does not guarantee that the program is executed with the DD value specified by the USING DATA pseudo instruction. Please note that no error messages from the assembler does not guarantee consistency. Final responsibility rests with the programmer.

(iv) Changing the DD's value

The DD's value can be reentered in three ways. Initially, the DD's value is reset directly by PSW access instructions when they are executed. Alternatively, DD is reentered automatically by the specific instructions that influence DD. DD can also be initialized by reset processing.

1. Directly Accessing PSW

Since DD is allocated to bit 12 of the PSW, it can be changed directly by PSW access instructions, the following are some of the practical examples.

ANDB PSWH, 11101111B ..... DD is reset, remaining bits unaffected

ORB PSWH, 00010000B ..... DD is set, remaining bits unaffected

XORB PSWH, 00010000B ..... DD is inverted, remaining bits unaffected

SB PSWH. 4 (or SB DD) ..... DD is set

RB PSWH. 4 (or RB DD) ... DD is reset

2. Specific Instructions Which Affect DD

When the following instructions are executed, DD is set automatically.

Commands that Set DD

L	A, obj	load instruction of word length for accumulator
MOV	A, obj	move instruction of word length for accumulator
CLR	A	clear instruction of word length for accumulator
POPS	A	pop (from system stack) instruction of word length for accumulator
EXTND	A	extend code instruction for accumulator

When the following instructions are executed, DD is reset automatically.

Commands that Reset DD

LB	A, obj	load instruction of byte length for accumulator
MOVB	A, obj	move instruction of byte length for accumulator
CLRB	A	clear instruction of byte length for accumulator

As shown in the above lists, the instructions that affect DD all transport the data from somewhere to the accumulator except for EXTND. When the concerned data is word long, DD is set, and DD is reset when the data is byte long. The instructions that are influenced by DD have been described in (iii). Each instruction in the group somehow operates on the accumulator. To operate on the accumulator, it is necessary to transfer the data from some place to the accumulator prior to the operation. For data which is transferred in word length, the accumulator will operate in word length. Similarly for data which is transferred in byte lengths, the accumulator will also operate in byte lengths. Thus, the matching of DD and the unit of the data is maintained.

3. Initialization by Reset Processing

DD is initialized or set to 0 by the reset processing. The reset processing is executed for the following

1. A proper reset pulse is applied to the $\overline{\text{RESET}}$ pin.

2. A BRK instruction is executed.

3. Overflow exists in WDT (watch dog timer).

«Caution: Setting DD at Program Interupts»

The interrupt procedure does not set or reset DD. DD is unrelated to the processes occuring following an interupt, hence the value of DD cannot be preddicted. When the interrupt procedure includes instructions that are affected by DD, it becomes necessary to initialize the flag. But the contents of the flag (namely

PSW) need not be saved, because they are saved automatically at the time of interrupt and returned by RTI (return from interrupt) instruction.

2.3 Concept of Segments

The concept of segments is used for managing the data of the assembler and the linker. As mentioned previously, the MSM66201 provides two memory spaces, the program and data memory spaces. When the programmer codes the assembler source program, he must tell the assembler in which memory space the entry is allocated. Then, the assembler employs segments that correspond to the physical memory space. Segments may be considered as logical spaces. The code segment of the assembler corresponds to the program space, while the data and bit segments correspond to the data space. For example, the program statement "Now, the following entry is to be saved in the program memory" becomes the assembler statement "Now the following entry is to be saved in the code segment". Thus the code segment is almost synonymous to the program memory.

The situation for the data memory is slightly different. The data memory is usually accessed using word or byte units, and occasionally a bit unit is used such as for setting flags. Such flag may be identified as "nth bit of address q" or "bit address n" which is more efficient in managing data (in the view of some programmers).

In the bit segment, bit 0 of byte 0 in the data memory is simply called the bit 0 address and the following bits are numerically sequenced in bit units as shown in Figure 1-9.

In the data segment, the data space is addressed in byte units, so the bit segment may seem to be created only to define the flags in an extreme sense.

The absolute segment is used for the entry of the data space of fixed address such as the vector table and VCAL table; it is also used when a certain data space is allocated to specific addresses for program convenience. The relocatable segment is used for the entry of the codes and the space that do not require specific addresses. The use of relocatable segment is recommended for programs in which many programmers are involved simultaneously, or for programs intended to be a submodule of a future program.

									Byte location
D	7	6	5	4	3	2	1	0	0
A	F	E	D	C	B	A	9	8	1
T	17	16	15	14	13	12	11	10	2
A	1F	1E	1D	1C	1B	1A	19	18	3
	27	26	25	24	23	22	21	20	4
M E M O									
R	7FFF7	7FFF6	7FFF5	7FFF4	7FFF3	7FFF2	7FFF1	7FFF0	FFFE
Y	7FFFF	7FFFE	7FFFD	7FFFC	7FFFB	7FFFA	7FFF9	7FFF8	FFFF
Bit location	7	6	5	4	3	2	1	0	

Figure 1-9. Bit Addresses

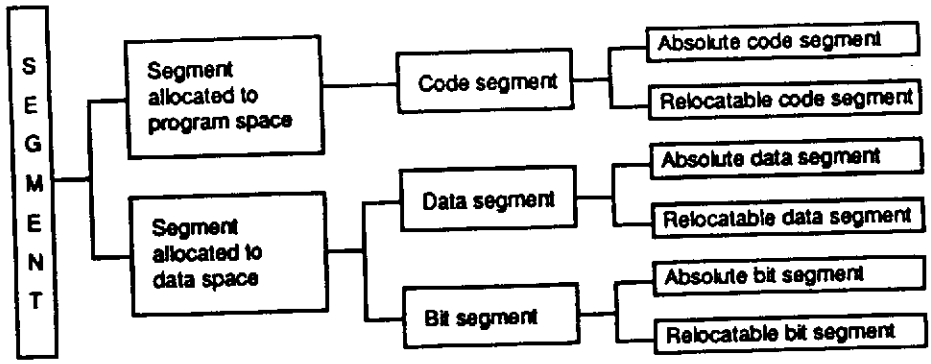


Figure 1-10 Segment Classification

The concept and outline of segments have been described so far. The following section describes the entry of segments to a real source program.

«Description of Absolute Segments»

For the assembler, three pseudo instructions are available to specify the segment type. Prior to the segment entry, these pseudo instructions: (CESG, DSEG, and BSEG) are entered to declare the segment type.

CSEG	This pseudo instruction declares that the following entries are to be located in the absolute code segment.
DSEG	This pseudo instruction declares that the following entries are to be located in the absolute data segment.
BSEG	This pseudo instruction declares that the following entries are to be located in the absolute bit segment

$$\left\{\begin{matrix}\text{CSEG}\\\text{DSEG}\\\text{BSEG}\end{matrix}\right\}\ \text{[AT starting address]}$$

The absolute code segment is used to enter the vector table, the VCAL table, and the ROM table, which is accessed by ROM reference instructions. The real statements entered will be labels, pseudo instructions which specify the data of word length (DW) and the data of byte length (DB), and the storage defining pseudo instruction (DS).

Refer to 2.1.1 for actual entry of statements.

When the relocatable assembler is used, the program main body (the instruction mnemonics of the MSM66201) is almost always entered in the relocatable code segment.

The actual entry of absolute data segment is explained here. The absolute data segment is entered to assign the data space to a specific addresses--for example, to assign the data space for global use to Page 0, or to allocate the stack space to certain fixed addresses. The entry of the absolute data segment is usually performed by the label and storage defining pseudo instruction (DS).

In Figure 1-11, the example entry assigns data of word length to WK_0, WK_1, WK_2, WK_3, and data of byte length to BUF_0, BUF_1, BUF_2, BUF_3 starting at the address $00C0_H$. When only the label values of the preceding example are desired to be defined, the DATA pseudo instruction may be used. The corresponding entry is shown in Figure 1-12. The latter entry differs from the former entry in that data space allocation is not possible for the latter. When the relocatable assembler is used with the latter entry, another relocatable segment might be allocated to the space behind $00C0_H$. This may be a concern,

while no such concern exists for the absolute assembler. DATA pseudo instruction is used for suballocation of the general data space or renaming of the label. An example of suballocation is shown in Figure 1-13. By placing the entry in Figure 1-13 after the entry in Figure 1-11, a data space of word length is suballocated to the low and high byte spaces.

```
DSEG    AT      00C0H
;
WK_0:   DS      2       ; Space for word long data
WK_1:   DS      2
WK_2:   DS      2
WK_3:   DS      2
;
BUF_0:  DS      1       ; Space for byte long data
BUF_1:  DS      1
BUF_2:  DS      1
BUF_3:  DS      1
        :
```

Figure 1-11. Absolute Data Segment

```
WK_0:   DATA    00C0H
WK_1:   DATA    WK_0+2
WK_2:   DATA    WK_0+4
WK_3:   DATA    WK_0+6
;
BUF_0:  DATA    0C08H
BUF_1:  DATA    BUF_0+1
BUF_2:  DATA    BUF_0+2
BUF_3:  DATA    BUF_0+3
        :
```

Figure 1-12. Specifying by DATA Pseudo Instruction

```
WK_0L   DATA    WK_0    ; Low byte of WK_0
WK_0H   DATA    WK_0+1  ; High byte of WK_0
WK_1L   DATA    WK_1    ; Low byte of WK_1
WK_1H   DATA    WK_1+1  ; High byte of WK_1
WK_2L   DATA    WK_2    ; Low byte of WK_2
WK_2H   DATA    WK_2+1  ; High byte of WK_2
WK_3L   DATA    WK_3    ; Low byte of WK_3
WK_3H   DATA    WK_3+1  ; High byte of WK_4
        :
```

Figure 1-13. Specifying by DATA Pseudo Instruction

As the last item described in this discussion on absolute segments, an example of a real program entry of bit segments is introduced.

The entry of a bit segment is performed by the label and bit long space defining pseudo instruction (DBIT), just as the entry of the data segment is done. The exact address arrangement of the bit segment is as shown in Figure 1-9. It is rather unwieldy to enter the address behind the AT portion of the BSEG pseudo instruction or the operand of the ORG pseudo instruction that controls the location counter directly with the bit addresses. For instance, bit 2 of byte address 1 is bit address 10. It is coded as 0001H.2, hence the following relationship exists.

address bit = address byte. n = address byte x 8 + n

An example of the entry of bit segments is described here. The following entry specifies the value of the four flags FLG_0, FLG_1, FLG_2, and FLG_3 starting at bit 0 of byte address $D0_H$.

```
            BSEG AT   00D0H.0
;
FLG_0:      DBIT      1
FLG_1:      DBIT      1
FLG_2:      DBIT      1
FLG_3:      DBIT      1
```

Figure 1-14. Entry of Absolute Bit Segment

In Figure 1-14, the first line statement "DSEG AT 00D0H.0" may be substituted by the statement "BSEG AT 0104".

The absolute bit segment may overlap partially or fully the absolute data segment. For example, the word long data space WK_0 is defined as the data segment; this is shown in Figure 1-13. For convenience, it became necessary for the lower four bits starting at WK_0 to be accessed using bit units, and the four bits are identified in the bit unit as WK_00, WK_01, WK_02 and WK_03. The new entry is easily made as shown in Figure 1-17 where the bit segment starts at WK_0.0. The assembler does not issue a warning for the overlapping condition.

There are two ways to specify the symbol which has the attribute of a bit segment. One way is to specify the label within the bit segment as shown in Figure 1-15, and the other way is to specify the label directly using a BIT pseudo instruction. The BIT pseudo instruction imparts values and the bit segment attribute to the symbol as shown in Figure 1-16.

```
            BSEG AT   WK_0.0
;
WK_00:      DBIT      1
WK_01:      DBIT      1
WK_02:      DBIT      1
WK_03:      DBIT      1
```

Figure 1-15. Bit Segment is Overlaid on Data Segment of Figure 1-12

```
WK_00:    BIT    WK_0.0
WK_01:    BIT    WK_0.1
WK_02:    BIT    WK_0.2
WK_03:    BIT    WK0.3
```

Figure 1-16. Equivalent Result (to Figure 1-15) Obtained by BIT Pseudo Instruction

[NOTE] The description of the coding formats and the functions of the assembler pseudo instructions given here is not complete because the main objective is to explain the concept of segments. For the details of the coding formats and the functions of the assembler pseudo instructions, refer to the assembler manual.

-Description of Relocatable Segments-

The entry of relocatable segments is described here, and it is more complex than the entry of the absolute segment. For the absolute segment, the entry is started by simply stating the pseudo instruction such as CSEG, DSEG or BSEG. For the relocatable segment, the name and type of segment are declared by the SEGMENT pseudo instruction, and even the type of relocation is declared if necessary. The entry format of the SEGMENT pseudo instruction is shown below.

```
segment name    SEGMENT    segment type    [relocation type]
```

The square bracket, [ ], for the relocation type implies that this entry in the statement is optional. This segment defining statement can be placed anywhere in the program as long as it precedes the RSEG pseudo instruction. However, it is usually placed at the head of program with other symbol defining statements.

The segment type specifies the segment to be defined: either code, data, or bit segment. The relocation type means the restricting factors when the linker allocates the segment to the memory addresses. The segment and relocation types are shown below.

Segment Type

CODE	defined segment is code segment
DATA	defined segment is data segment
BIT	defined segment is bit segment

Relocation Type

UNIT	match segment head with unit boundary
WORD	match segment head with word boundary
OCT	match segment head with 8-byte boundary
PAGE	match segment head with page head
NUMBERn	match segment head with boundary of n units where n=1, 2, 4, 8, . . . 2048 (power of 2)
INPAGE	match segment head within page

In the following description, the unit means the smallest unit of segment used in the accessing operation involved; hence the unit for the code and data segments is a byte, while it is a bit for the bit segment.

The UNIT in the relocation type above means "allocate any place". Without declaration of the relocation type listed in the above table, the assembler assumes that the segment possesses the UNIT attributes and processes the segment accordingly. Therefore, UNIT is not usually used unless its explicit understanding is required. As it is easily understood from the preceding discussions, the attributes of WORD and OCT concern the word boundary and the local register accordingly.

PAGE and INPAGE obviously take into account the page structure of the data space. NUMBER n permits the specification and allocation of boundaries which can not be defined by WORD, OCT and PAGE. Number, in this case, means so many units, and the unit is byte for code and data segments and bit for the bit segment. The attributes of UNIT, WORD, OCT and PAGE are inclusive in ascending order. For example, specifying OCT (its attributes) necessarily specifies all of the attributes of UNIT and WORD, and specifying PAGE satisfies all of the attributes of UNIT, WORD and OCT.

To enter the relocatable segment statement, the RSEG pseudo instruction whose format is shown below is used. This starts the entry of the segment indicated in the format.

RSEG	segment name

Following the entry of RSEG pseudo instruction, each segment is entered in the same manner as for the absolute segment, so no further description is given. It should be noted that overlapping the data and bit segments is not allowed for relocatable segments, while a similar overlapping for absolute segments is permissible.

For reference, an example entry is given in the following pages for an actual case where the absolute and relocatable segments coexists. Following this example, another important concept for the relocatable segment, "partial segment", will be explained.

(Example of Segment Entry)

```
            MAIN        SEGMENT   CODE        ; Main Routine
            MAIN_BUF    SEGMENT   DATA WORD   ; Buffer Area for Main Routine
            MAIN_FLG    SEGMENT   BIT         ; Flag Area for Main Routine
;
            STACK_SIZE  EQU       256         ; Stack Size
;
;           ***** Definition of Stack Space *****
            DSEG    AT    0280H-STACK_SIZE    ; Reserve a stack space with a size of
            DS      STACK_SIZE-1              ; STACK_SIZE starting at address 0280H
STACK_BTM   DS      1
;
;           ***** Vector Table *****
;           CSEG    AT    00C0H.0
            DW      START                     ; Reset vector
            DS      38                        ; Reserved for future
;
;           ***** VCAL Table *****
            CSEG    AT    0028H
            DS      16                        ; Reserved for future
;
;           ***** Flag Definition *****
            BSEG    AT    00C0H.0
FLG0:       DBIT    1
FLG1:       DBIT    1
FLG2:       DBIT    1
FLG3:       DBIT    1
            DBIT    4                         ; Reserved for future
;
;           ***** ROM Table *****
            CSEG    AT    3C00H
TBL_1:      DW      56DCH, 0013H, 9FA4H, 0A37H
TBL_2       DB      03H, 05H, 5FH, 87H, 0C4H, 0A1H
;
;           ***** WORK *****
            DSEG    AT    00C1H
WK_0:       DS      2
WK_1:       DS      2
WK_2:       DS      2
WK_3:       DS      2
;
```

(Continued on next page)

```
;           **********************
;           *  MAIN ROUTINE      *
;           **********************
            RSEG    MAIN

START:
            Main Routine Program Section

            RSEG    MAIN_BUF
            Main Routine Data Section

            RSEG    MAIN_FLG
            Main Routine Flag Section

                    :

            END
```

«Partial Segments»

Sometimes, when creating program code, relocatable segments are referred to as partial segments. Here is an example. Imagine the situation in which three programmers, A, B, and C, are in the process of designing a program as a team. Assume that a module in the program calls six subroutines which are named SUB1, SUB2, SUB3, SUB4, SUB5 and SUB6, respectively. Programmer A is assigned to SUB1 and SUB2, Programmer B to SUB3 and SUB 4, and Programmer C is assigned to SUB5 and SUB6. Each programmer works on his assigned subroutines independently using three different files. The completed files are assembled independently and the resulting relocatable objects are linked by the linker. The six subroutines, SUB1 through SUB6, are lumped in the code segment named SUBROOT that will be allocated to a continuous area of the memory space. Then, to implement the preceding plan, the programmers enter their statements in the source programs in the following manner.

1. Define the relocatable code segment named SUBROOT by SEGMENT pseudo instruction.
2. Each programmer enters RSEG SUBROOT prior to entering SUB1 through SUB6.

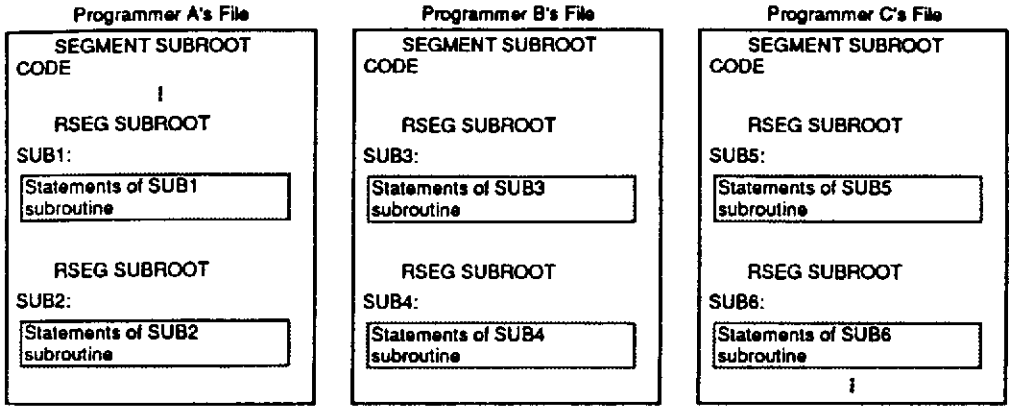


Figure 1-17. Example of Partial Segment Coding

The files, some of which are in shown in Figure 1-17, are assembled by the assembler RAS66K, and each of the resulting three object files contain two of the relocatable segments named SUBROOT. When the three object files are linked, the linker will find six relocatable segments of the identical name; however, the linker will consider them as one entity of SUBROOT, and allocate the six segments to a continuous memory space. Each of the six segments prior to the last allocation by the linker is called a partial segment. Each partial segment:

- is a relocatable segment
- has an identical segment name
- has identical segment attributes
- has identical relocation attributes
- implies the existence of plural segments

Among relocation attributes, the UNIT attribute should agree with all other attributes.

The linker regards the partial segments as the subdivisions of one larger segment whenever they are found, and allocates them to a continuous space.

Chapter 2. Addressing Modes

The MSM66201 provides two independent memory spaces, namely the program and data memory spaces. The MSM66201 accommodates the accessing of these memory spaces through a number of different addressing modes. The program memory space usually consists of ROM and is referred to as ROM space in this chapter. The data memory space generally consists of RAM and is referred to as RAM space. The RAM space includes registers, counters, ports and other pieces belonging to SFR, as well as the pointing and local registers. Addressing the RAM and ROM is referred to as RAM addressing and ROM addressing respectively.

1. RAM Addressing Modes

Except for a few cases, RAM addressing modes are generally used for the operands of the instructions. There are five major classes of RAM addressing modes, which are listed below:

1. Register addressing
2. Page addressing
3. Pointing register indirect addressing
4. Stack addressing
5. Immediate addressing

1.1 Register addressing

Register addressing, for accessing the contents of the registers, includes the following:

(1) Accumulator addressing

[Symbol]	A

The above is the addressing mode for the accumulator. It is applicable equally to word long and byte long data.

Example:

```
L     A, #0        LB    A, #0
MOV   A, er0       MOVB  A, #0
```

(2) Pointing register addressing

[Symbol]	DP	(data pointer)
	USP	(user stack pointer)
	X1, X2	(index register)

This addressing mode is for pointing registers, handling the registers individually. This addressing mode can be used in word length instruction only.

[NOTE] There are eight units of pointing registers (PR0 through PR7), and the units to be accessed by These modes are specified by the system control base (SCB).

Example:
```
L     A, DP          MOV   USP, A
MOV   X1, #2000H
ROL   X2
```

(3) Control register addressing

[Symbol]	LRB	(local register base)
	PSW	(program status word)
	PSWH	(program status word upper byte)
	PSWL	(program status word lower byte)

Above, the addressing modes for the local register base and program status word are shown. LRB and PSW are applicable only to word long data, while PSWH and PSWL are applicable only to byte long data.

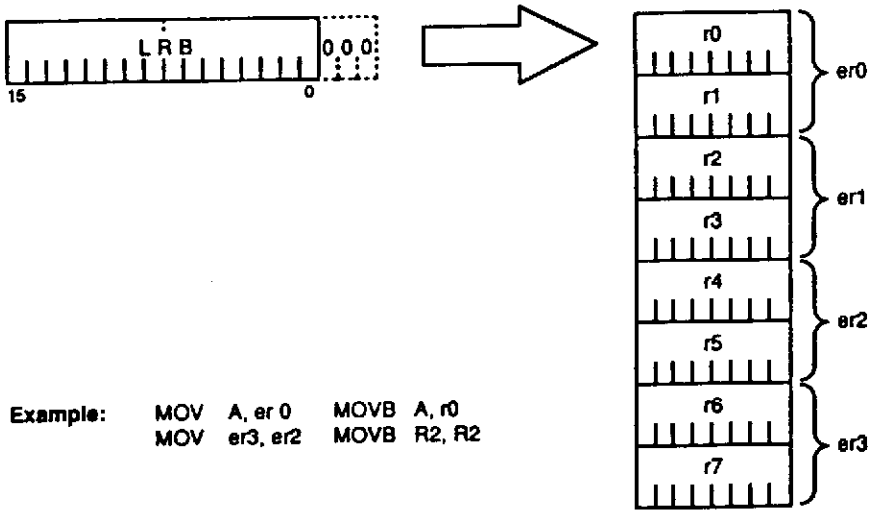
Example:
```
L     A, LRB         MOV   PSW, #00H
MOVB  PSWH, A        ROLB  PSWL
```

(4) Local register addressing

[Symbol]	r0—r7	(local register addressing)
	er0—er3	(extended local register addressing)

Above, the addressing modes for local registers are shown. The base address of the set of the local registers is generated by adding three bits of 0 to the contents of LRB (local register base).

Furthermore, r0—r7, and er0—er3, may respectively be stated as R0—R7, and ER0—ER3.



Example:
```
MOV   A, er 0        MOVB  A, r0
MOV   er3, er2       MOVB  R2, R2
```

(5) System stack pointer addressing

[Symbol]	SSP

Above, the addressing mode for the system stack pointer is shown. It is applicable only to word long data.

Example: MOV A, SSP MOV SSP, #1000H

1.2 Page addressing

In this addressing mode, an address is specified by offsetting from the starting address of the page. There are two modes of page addressing: current page addressing and zero page addressing modes. In current page addressing, the offset is given from the starting address of the page indicated by LRB (local register). In zero page addressing, the offset within Page 0 is not affected by LRB.

(1) Current page addressing

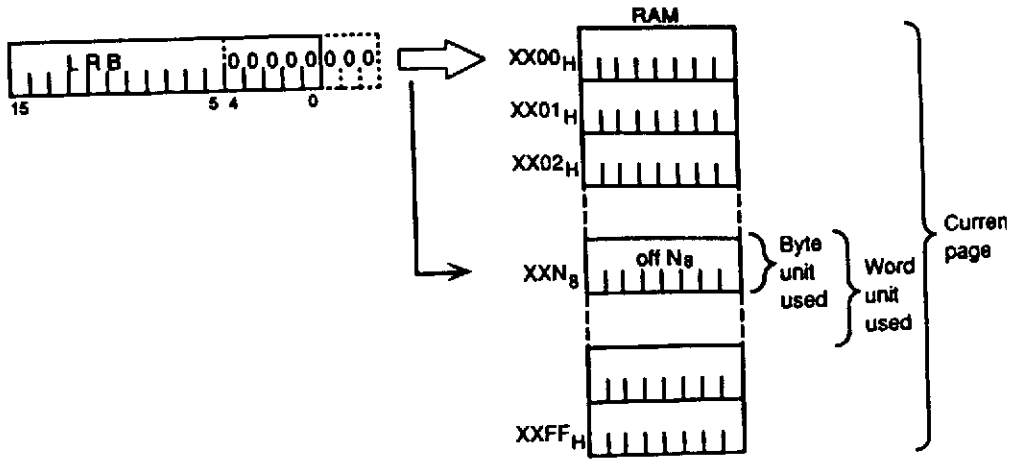
[Symbol]	off*1 N_8*2

As shown in the diagram, bits 12 through 5 of LRB indicate the page address in RAM space. Bits 4 through 0 of LRB with an additional 3 bits form the address (8 bits) within the page. All 8 bits are 0 for the base address of the page, and the offset (0 ~ 255) is given from the base address.

This addressing mode is applicable to word long data and byte long data.

[NOTE] 1 "off" is the agreed code to specify current page addressing. It may be written as "OFF". However, the latter expression resembles 0FFH for the immediate value. To avoid possible confusion, the use of "off" is recommended.

2 N_8 is the amount of the offset within the page. However, at the coding of the source program, the absolute, non-offset address itself or a symbol such as the label is entered. The programmer should not be concerned with offsetting.



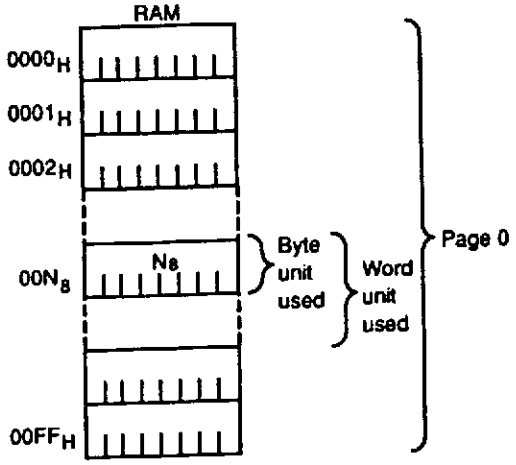
Example:

```
L     A, off 2000H      MOV   A, off WORK_01
LB    A, off 2000H      MOVB  A, off WORK_01
```

(2) Zero Page addressing

[Symbol]	N8

Above is the zero page addressing mode, where the offset is given within Page 0 (addresses 0 through $0FF_H$) in the RAM space. This addressing mode is applicable to word long and byte long data.



[NOTE] SFR is located in Page 0. To simplify the entry of addresses to SFR, the assembler defines the data address symbols, some of which are easily mistaken as one of the special assembler symbols used to describe the addressing modes. Care should be observed. For instance, A is a special assembler symbol used to describe addressing modes such as ACC for zero page addressing, and off ACC for current page addressing.

More examples of confusing symbols are listed below.

Example 1:	MOV	PSW, A	register addressing
	MOV	APSW, A	zero page addressing
	MOV	off APSW, A	current page addressing
Example 2:	MOV	LRB, A	register addressing
	MOV	ALRB, A	zero page addressing
	MOV	off ALRB, A	current page addressing
Example 3:	MOV	SSP, A	register addressing
	MOV	ASSP, A	zero page addressing
	MOV	off ASSP, A	current page addressing
Example 4:	MOVB	PSW, A	register addressing
	MOVB	APSW, A	zero page addressing
	MOVB	off APSW, A	current page addressing

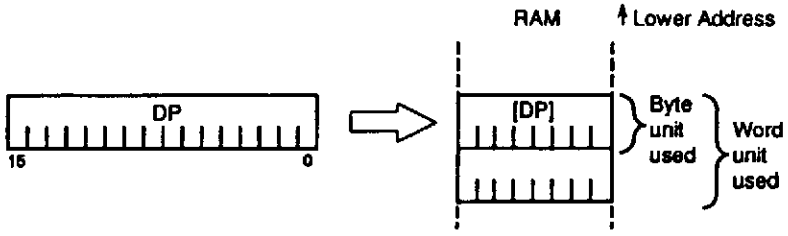
1.3 Pointing register indirect addressing

This addressing mode is performed by the pointing registers using indirect addressing, and it is used for both word and byte length instructions.

(1) Data pointer indirect addressing

[Symbol]	[DP]

The RAM space whose address is specified by the contents of the data pointer is accessed by this mode.



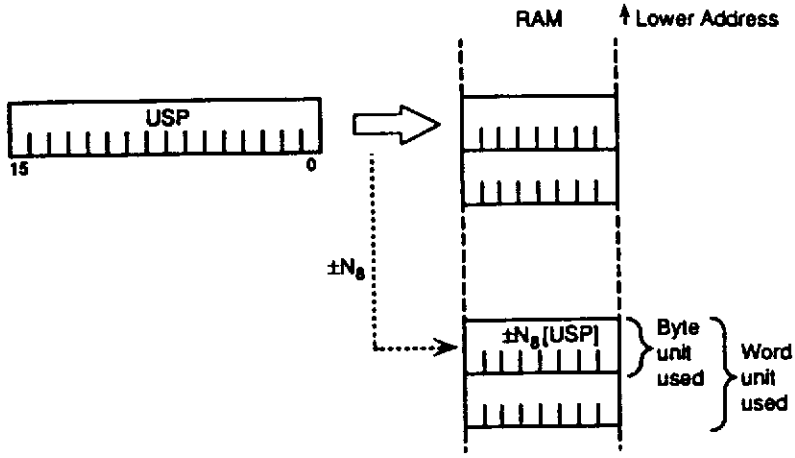
Example:

```
L     A, [DP]
LB    A, [DP]
```

(2) User stack pointer indirect addressing (with 8-bit displacement)

[Symbol]	±N8[USP]

The RAM space whose address is specified by the sum of the user stack pointer contents and $\pm N_8$ (displacement) is accessed. $\pm N_8$ is in the range from +127 to -128 or the equivalent range specified by symbols.



[NOTE] In the machine code, $\pm N_8$ is the displacement code sign which is given by the uppermost bit; however, the integer between +127 and -128 itself is used at program entry. For instance, the code for the displacement of -1 is FFH and the entry seems to be 0FFH[USP]; however, it is incorrect for the assembler, because its value exceeds +127. The correct entry in this case is -1 [USP].

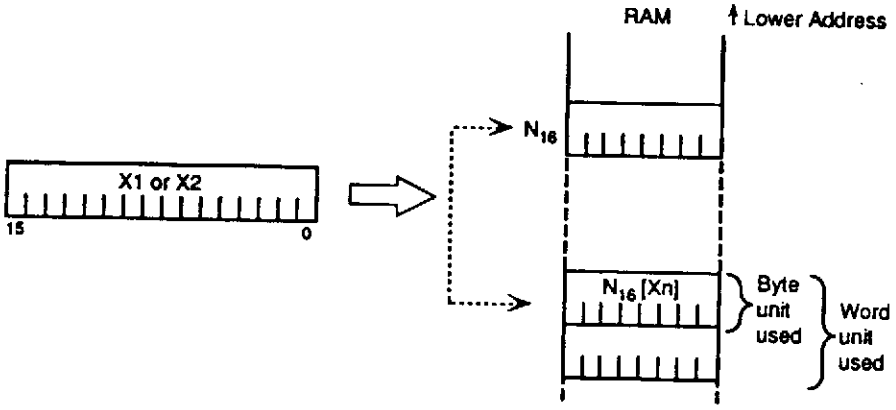
```
Example:    L     A, 70H[USP]        LB    A, -120[USP]
            ADD   A, DISP1[USP]
```

(3) Index register indirect addressing (with 16-bit base)

[Symbol]	N_{16}[X1], N_{16}[X2]

The RAM space whose address is the sum of the base address N_{16} (16 bits) and the contents of the index register is accessed.

The addition of N_{16} and X1, or N_{16} and X2 is performed in 16 bit length without sign, and the overflow is ignored. The maximum space addressable is 64K bytes; namely, the entire space of the *current bank*.



Example: L A, 5000H[X1] LB A, TBL_BASE[X2]

1.4 Immediate addressing

[Symbol]	#N_8, #N_{16}

The number or symbol in the operand itself specifies the address. The operand statement is #N_{16} for word long data and #N_8 for byte long data.

Example: MOV er0, #1234H MOVB r0, #12H

«Advice About RAM Addressing»

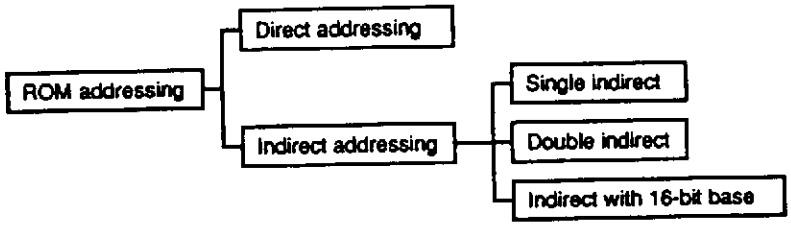
In RAM addressing modes, there is not a 64K-space direct addressing mode. The design of the CPU is based on the idea of processing a job completely within a page of 256 bytes. The data area required for the job is contained in the page as much as possible so that direct page addressing (off N8) can be used.

On the other hand, some data area need to be accessed by many processes. For example, a global variable area may need to be accessed by multiple subroutines. Special function registers (SFR) are another good example. This is why SFR is allocated to Page 0, and usually accessed by zero page addressing. Also, allocating a global variable area to Page 0 enables the use of zero page addressing regardless of the value of the local register (LRB).

When direct accesses are desired beyond the page limit of 256 bytes, the index-register indirect addressing with a 16-bit base (N_{16}[X1], N_{16}[X2]) is used. X1 and X2 are 0 while N_{16} is the address of the object in this case. However, the preceding addressing is not efficient because of the large numbers of bytes required for the instruction and the number of cycles it uses. Current page addressing used with careful data arrangement or register addressing with the local registers strategically placed in the areas should be employed as much as possible for better program efficiency.

2. ROM Addressing Modes

ROM addressing is used for accessing program memory space which consists of on-chip ROM and external program memory. ROM addressing is classified into two types, the direct and the indirect type. The indirect addressing type is further classified into three sub-types: single indirect addressing, double indirect addressing, and indirect addressing with a 16-bit base.



The ROM addressing modes are specified only in the operands of the ROM table reference instructions such as LC, LCB, CMPC, CMPCB.

The addressing modes are described below.

2.1 Direct addressing

[Symbol]	N^{*}_{16}

The reference address is directly specified with 16-bit immediate addressing N^{*}_{16}.

Example:

```
LC      A, 2000H
LC      A, DATA_TABLE
LCB     A, 3800H
```

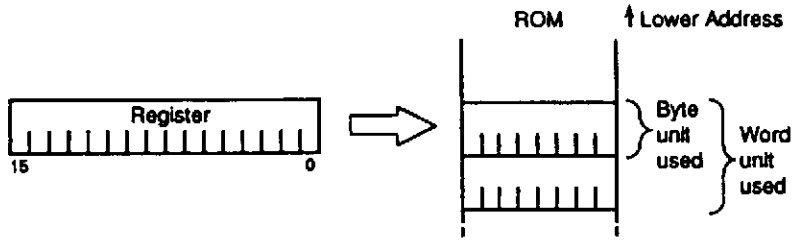
2.2 Indirect addressing

Indirect addressing includes: single indirect addressing, in which the register and the contents of the RAM specifies the object address; double indirect addressing, in which the contents of the RAM addressed by the pointing register specifies the object address; and indirect addressing with a 16-bit base.

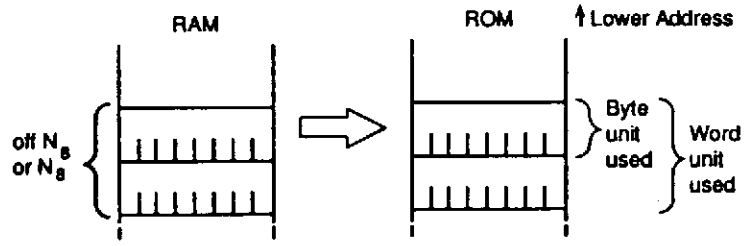
(1) Single indirect addressing

[Symbol]		
	1.	local register indirect ..... [er0], [er1], [er2], [er3]
	2.	pointing register indirect ..... [DP], [X1], [X2], [USP]
	3.	SSP indirect ..... [SSP]
	4.	LRB indirect ..... [LRB]
	5.	RAM indirect ..... [off N_8], [N_8]

For cases 1—4, the contents of each 16-bit register (er0—er3,DP, X1, X2, USP, SSP, LRB) is the address of the ROM to be accessed.



For case 5, the contents (16 bits) of the RAM which is specified by current or zero page addressing in word length is the address of the ROM to be accessed.



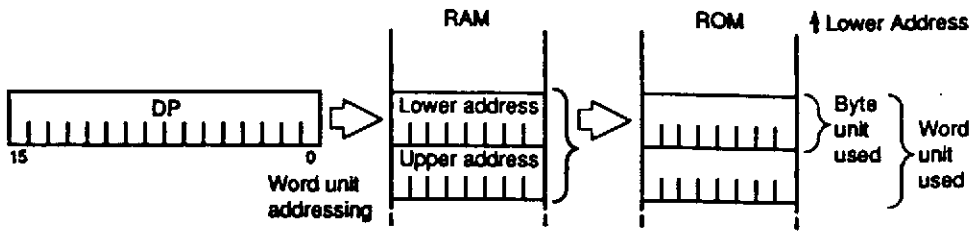
Example:

```
LC    A, [er0]          LC    A, [DP]
LC    A, [SSP]          LCB   A, [LRB]
LCB   A, [off 20H]      LC    A, [off DATA_TBL]
LCB   A, [ACC]
```

(2) Double indirect addressing

The section below describes double indirect addressing, in which the RAM addressed indirectly by the pointing register specifies the ROM indirectly.

[Symbol]		
	1. DP double indirect	[[DP]]
	2. USP double indirect (with signed 8-bit displacement)	[±N_8[USP]]
	3. Index register double indirect (with 16-bit base)	[N_{16}[X1]],[N_{16}[X2]]



Example:

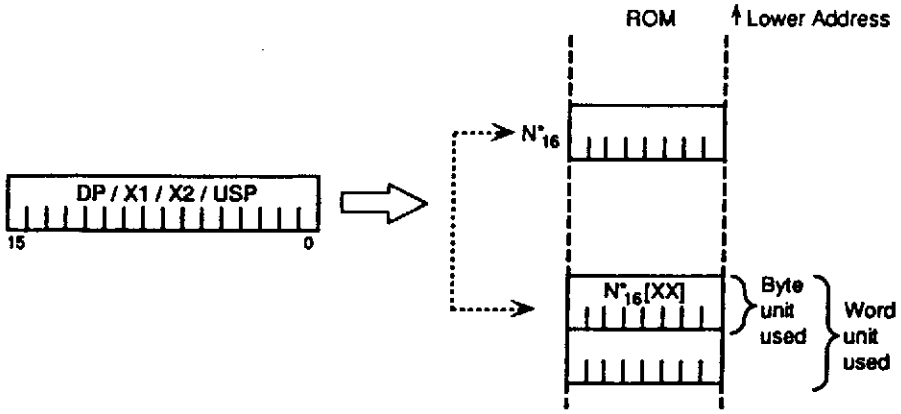
```
LC    A, [[DP]]
LC    A, [-10[USP]]
LCB   A, [1000H[X1]]
LC    A, [OFFSET_1[X2]]
```

(3) Indirect addressing with 16-bit base

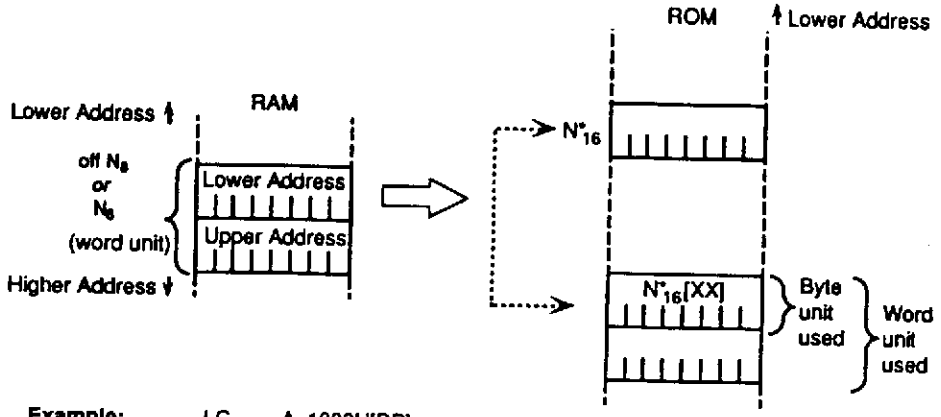
[Symbol]		
	1. pointing register indirect	N°_{16}[DP], N°_{16}[X1] N°_{16}[X2], N°_{16}[USP]
	2. RAM indirect	N°_{16}[off N_8], N°_{16}[N_8]

The sum of the immediate address (N°_{16}, the base), and the contents of the pointing register or the RAM (word long) is the address of the object.

Pointing Register Indirect

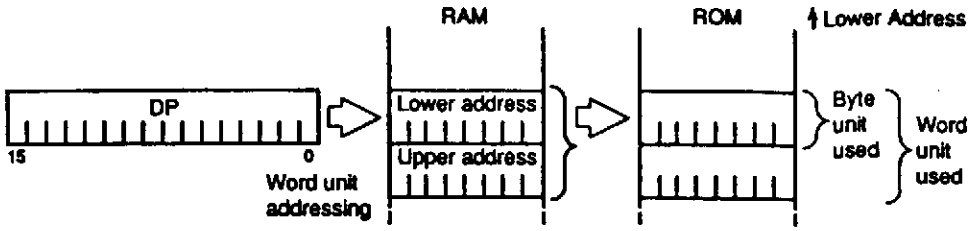


RAM Indirect



Example:

```
LC    A, 1000H[DP]
LC    A, EXT_ROM_BASE[X1]
LC    A, OFFSET_1[off DISP]
```



Example:

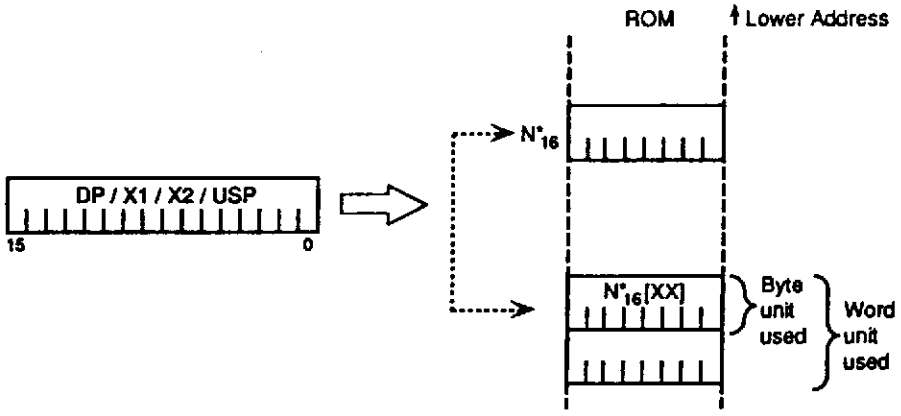
```
LC    A, [[DP]]
LC    A, [-10[USP]]
LCB   A, [1000H[X1]]
LC    A, [OFFSET_1[X2]]
```

(3) Indirect addressing with 16-bit base

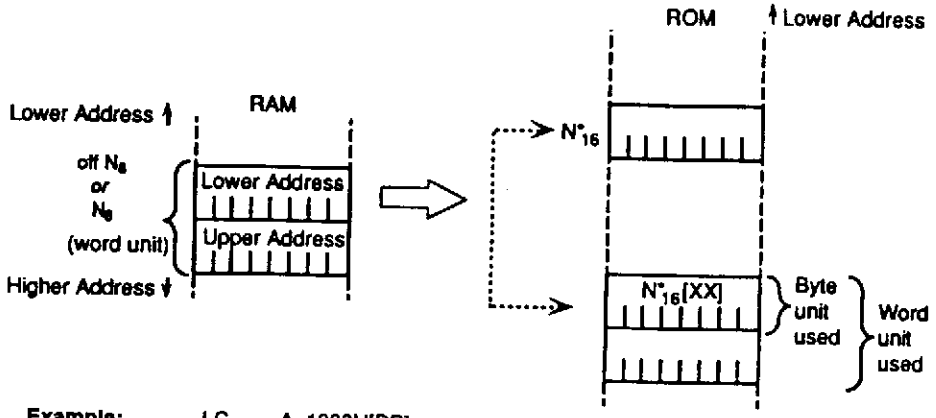
[Symbol]		
	1. pointing register indirect	N^{*}_{16}[DP], N^{*}_{16}[X1] N^{*}_{16}[X2], N^{*}_{16}[USP]
	2. RAM indirect	N^{*}_{16}[off N_8], N^{*}_{16}[N_8]

The sum of the immediate address (N^{*}_{16}, the base), and the contents of the pointing register or the RAM (word long) is the address of the object.

Pointing Register Indirect



RAM Indirect



Example:

```
LC    A, 1000H[DP]
LC    A, EXT_ROM_BASE[X1]
LC    A, OFFSET_1[off DISP]
```

3. Bit Addressing Modes

The addressing instructions of the MSM66201 includes those which permit accessing a specific bit desired. Examples of these are SBR and SB. The addressing of the desired bit is realized by combining the locations of byte and bit. The byte location is specified by RAM addressing. The bit location is specified either by the bits from 0 to 2 of the accumulator indirectly or by the operand directly. Note that usable RAM addresses are byte long in length.

(1) Specifying a bit using the accumulator indirectly

This mode appears in the SBR, TBR, and MBR instructions. Only the following RAM addressing modes are available to specify the byte location.

r0—r7, PSWH, PSWL, off N_8, N_8, [DP], $\pm N_8$[USP], N_{16}[X1], N_{16}[X2]

Example:

```
SBR     r0              RBR     PSWH
TBR     off FLAGS       MBR     C, [DP]
```

(2) Specifying the bit directly

This mode appears in the SB, RB, and JRB instructions. Addressing is accomplished by specifying the object bit with the bit location (from 0 to 7) and byte location specified by the RAM addressing described in (1).

In the statement (shown below), the RAM addressing entry and the bit location entry are connected with a dot (.).

[Symbol] RAM addressing code.Bit location

Example:

```
SB      r0.2
RB      PSWH.BIT_POINT
JBR     off FLAGS.0, TIMER_LOOP
```

4. Logical Bit Address Space

Bit addressing for the MSM66201 is performed by specifying the bit address with the byte location specified by RAM addressing, and the bit location within the byte thus specified.

There is no physical bit address space for the MSM66201; however, its use for the assembler statement is acknowledged. Because the bit addressing statement with a combination of byte and bit locations is permitted for assemblers of the MSM80C51 series, such bit address statements are also recognized by the MSM66201 assembler. Strictly speaking, the bit address space is a logical concept, and its realization at the assembler level is accomplished by combining byte and bit locations.

The logical bit addresses of the MSM66201 start at address 0 (that is, location 0 of the data memory) and they progress in ascending order: address 1, address 2, and so on.

Logical Bit Address

									Byte location
DATA MEMORY	7	6	5	4	3	2	1	0	0
	F	E	D	C	B	A	9	8	1
	17	16	15	14	13	12	11	10	2
	1F	1E	1D	1C	1B	1A	19	18	3
	27	26	25	24	23	22	21	20	4
	7FFF7	7FFF6	7FFF5	7FFF4	7FFF3	7FFF2	7FFF1	7FFF0	FFFE
	7FFFF	7FFFE	7FFFD	7FFFC	7FFFB	7FFFA	7FFF9	7FFF8	FFFF
Bit location	7	6	5	4	3	2	1	0	