KEIL SOFTWARE

A51 Assembler
A251 Assembler

Macro Assemblers for the 8051 and MCS® 251 Microcontrollers

User's Guide 04.95

A15 D05/12/95

Preface

This manual describes how to use the A51 and A251 macro assemblers. The A51 and A251 assembler translate programs you write in assembly language into executable machine instructions. You may use the A51 assembler to assemble programs for the 8051 family of microcontrollers. You may use the A251 assembler to assemble programs for the 8051 family as well as the MCS 251 family of microcontrollers. This manual assumes that you are familiar with the MS-DOS operating system and know how to program the 8051 or MCS 251 microcontrollers.

This manual is divided into the following chapters.

"Chapter 1. Introduction," describes the basics of assembly language programming.

"Chapter 2. 8051 and MCS 251 Architecture," contains an overview of the 8051 and MCS 251 hardware.

"Chapter 3. Writing Assembly Programs," describes assembler statements, operands and address descriptors, and the rules for arithmetic and logical expressions.

"Chapter 4. Assembler Directives," describes how to define segments and symbols and how to use all directives.

"Chapter 5. Standard Macros," describes the function of the standard macros and contains information for using standard macros.

"Chapter 6. Macro Processing Language," defines and describes the use of the Intel Macro Processing Language.

"Chapter 7. Invocation and Controls," describes how to invoke the assembler and how to control the assembler operation.

"Chapter 8. Error Messages," contains a list of all assembler error messages and describes their causes and how to avoid them.

The Appendix includes information on the 8051 and MCS 251 instruction set, a summary of directives and controls, the differences between assembler versions, and other items of interest.

Document Conventions

This document uses the following conventions:

Examples	Description
README.TXT	Bold capital text is used for the names of executable programs, data files, source files, environment variables, and commands you enter at the MS-DOS command prompt. This text usually represents commands that you must type in literally. For example: **CLS** **DIR** **BL51.EXE** Note that you are not required to enter these commands using all capital letters.
`Courier`	Text in this typeface is used to represent information that displays on screen or prints at the printer. This typeface is also used within the text when discussing or describing command line items.
Variables	Text in italics represents information that you must provide. For example, *projectfile* in a syntax string means that you must supply the actual project file name. Occasionally, italics are also used to emphasize words in the text.
Elements that repeat…	Ellipses (…) are used in examples to indicate an item that may be repeated.

Omitted code
.
.
.
	Vertical ellipses are used in source code examples to indicate that a fragment of the program is omitted. For example:
	`void main (void) {`
	`.`
	`.`
	`.`
	`while (1);`

⟦*Optional Items*⟧	Optional arguments in command-line and option fields are indicated by double brackets. For example: **`C51 TEST.C PRINT ⟦(filename)⟧`**
{ *opt1* | *opt2* }	Text contained within braces, separated by a vertical bar represents a group of items from which one must be chosen. The braces enclose all of the choices and the vertical bars separate the choices. One item in the list must be selected.
Keys	Text in this sans serif typeface represents actual keys on the keyboard. For example, “Press **Enter** to continue.”

Contents

Chapter 1. Introduction

This manual describes the A51 macro assembler and the A251 macro assembler and explains the process of developing software in assembly language for the MCS 251 and 8051 microcontroller families.

A brief overview of the 8051 and MCS 251 architecture can be found in "Chapter 2. 8051 and MCS 251 Architecture" on page 9. In this overview, the differences between the generic 8051 and the MCS 251 processors are described. For the most complete information about the 8051 or MCS 251 microcontrollers, contact your vendor.

Assembly language programs translate directly into machine instructions which instruct the processor what operations to perform. Therefore, to effectively write assembly programs, you should be familiar with both the microcomputer architecture and assembly language. This chapter presents an overview of the A251 macro assembler and how it is used.

The A251 assembler is a superset of A51 assembler. For this reason, this manual serves as documentation for both assemblers. The term A251 is used within this document to refer to both the A251 assembler and A51 assembler.

NOTE
New features in the A251 assembler and in the MCS 251 microcontroller family which are not available in the A51 assembler or the 8051 microcontroller family are marked with †.

What is an Assembler?

An assembler is a software tool – a program – designed to simplify the task of writing computer programs. It performs the clerical task of translating symbolic code into executable object code. This object code may then be programmed into an 8051 or MCS 251 microcontroller and executed. If you have ever written a computer program directly in machine-recognizable form, such as binary or hexadecimal code, you will appreciate the advantages of programming in symbolic assembly language.

Assembly language operation codes (mnemonics) are easily remembered (MOV for move instructions, ADD for addition, and so on). You can also symbolically

1

express addresses and values referenced in the operand field of instructions. Since you assign these names, you can make them as meaningful as the mnemonics for the instructions. For example, if your program must manipulate a date as data, you can assign it the symbolic name DATE. If your program contains a set of instructions used as a timing loop (a set of instructions executed repeatedly until a specific amount of time has passed), you can name the instruction group TIMER_LOOP.

An assembly program has three constituent parts:

- Machine instructions
- Assembler directives
- Assembler controls

A machine instruction is a machine code that can be executed by the machine. Detailed discussion of the machine instructions can be found in the hardware manuals of the 8051 or MCS 251 microcontrollers. Appendix A provides an overview about machine instructions.

Assembler directives are used to define the program structure and symbols, and generate non-executable code (data, messages, etc.). Refer to “Chapter 4. Assembler Directives” on page 41 for details on all of the assembler directives.

Assembler controls set the assembly modes and direct the assembly flow. “Chapter 7. Invocation and Controls” on page 113 contains a comprehensive guide to all the assembler controls.

How to Develop A Program

The A251 assembler enables the user to program in a modular fashion. The following paragraphs explain the basics of modular program development.

Advantages of Modular Programming

Many programs are too long or complex to write as a single unit. Programming becomes much simpler when the code is divided into small functional units. Modular programs are usually easier to code, debug, and change than monolithic programs.

The modular approach to programming is similar to the design of hardware that contains numerous circuits. The device or program is logically divided into "black boxes" with specific inputs and outputs. Once the interfaces between the units have been defined, the detailed design of each unit can proceed separately.

Efficient Program Development

Programs can be developed more quickly with the modular approach since small subprograms are easier to understand, design, and test than large programs. With the module inputs and outputs defined, the programmer can supply the needed input and verify the correctness of the module by examining the output. The separate modules are than linked and located by the linker into an absolute executable single program module. Finally, the complete module is tested.

Multiple Use of Subprograms

Code written for one program is often useful in others. Modular programming allows these sections to be saved for future use. Because the code is relocatable, saved modules can be linked to any program which fulfills their input and output requirements. With monolithic programming, such sections of code are buried inside the program and are not so available for use by other programs.

Ease of Debugging and Modifying

Modular programs are generally easier to debug than monolithic programs. Because of the well defined module interfaces of the program, problems can be isolated to specific modules. Once the faulty module has been identified, fixing the problem is considerably simpler. When a program must be modified, modular programming simplifies the job. You can link new or debugged modules to an existing program with the confidence that the rest of the program will not change.

Modular Program Development Process

This section is a brief discussion of the program development process with the relocatable A251 assembler, L251 Linker/Locator, and the OH251 code conversion program.

1

Segments, Modules, and Programs

In the initial design stages, the tasks to be performed by the program are defined, and then partitioned into subprograms. Here are brief introductions to the kinds of subprograms used with the A251 assembler and L251 linker/locator.

A segment is a block of code or data memory. A segment may be relocatable or absolute. A relocatable segment has a name, type, and other attributes. Segments with the same name, from different modules, are considered part of the same segment and are called "partial segments". Partial segments are combined into segments by L251. An absolute segment cannot be combined with other segments.

A module contains one or more segments or partial segments. A module has a name assigned by the user. The module definitions determine the scope of local symbols.

A program consists of a single absolute module, merging all absolute and relocatable segments from all input modules.

Program Entry and Exit

After the design is completed, the source code for each module is entered into a disk file using any text editor. When errors are detected in the development process, the text editor may be used to make corrections in the source code.

Assembly

The A251 assembler translates the source code into object code. The assembler produces a relocatable object file and a listing file showing the results of the assembly. When the assembler invocation contains the **DEBUG** control, the object file also receives the debug information for use during the symbolic debugging of the program. This debugging may be via the dScope-251 Debugger/Simulator, or in-circuit emulators available from many vendors.

Object File: the object file contains machine language instructions and data that can be loaded into memory for execution or interpretation. In addition, it contains control information governing the loading process.

Listing File: The listing file provides both the source program and the object code. The assembler also produces diagnostic messages in the listing file for syntax and other coding errors. For example, if you specify a 16-bit value for an instruction that can only use an 8-bit value, the assembler tells you that the value exceeds the permissible range. Appendix F describes the format of the listing file. In addition, you can also request a symbol table to be appended to the listing. The symbol table lists all the symbols and their attributes.

Relocation and Linkage

After assembly of all modules of the program, L251 processes the object module files. the L251 program assigns absolute memory locations to all the relocatable segments, combining segments with the same name and type. L251 also resolves all references between modules. L251 outputs an absolute object module file with the completed program, and a summary listing file showing the results of the link/locate process.

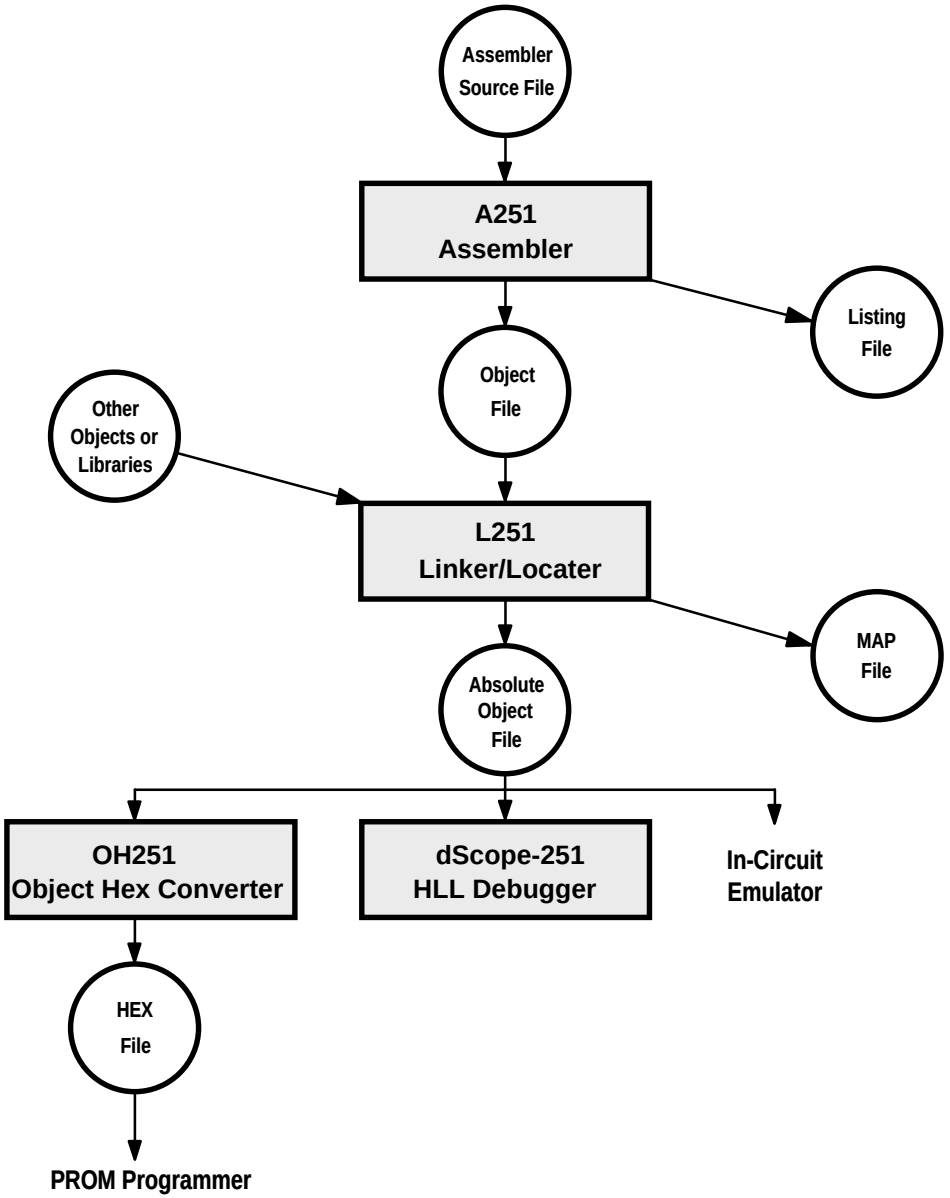
Keeping Track of Files

It is convenient to use the extensions of filename to indicate the stage in the process represented by the contents of each file. Thus, source code files can use extensions like .SRC or .A51 (indicating that the code is for input to the A251 assembler). Object code files receive the extension .OBJ by default, or the user can specify another extension. Executable files generally have no extension. Listing files can use .LST, the default extension assigned by the assembler. L251 uses .MAP for the default linker map file extension. L51 and BL51 use .M51 for the default linker map file extension.

1

Writing and Assembling Programs

There are several steps necessary to incorporate an 8051 microcomputer in your application. The following figure shows an overview of the steps involved in creating a program for the 8051 or 251.



If you are developing hardware for your application, consult the 8051, MCS 51, or MCS 251 hardware manuals.

Following is an example listing file generated by the assembler.

```
A251 MACRO ASSEMBLER  ASSEMBLER DEMO PROGRAM            24/11/94 10:09:15 PAGE     1

DOS MACRO ASSEMBLER A251 V1.00
OBJECT MODULE PLACED IN DEMO.OBJ
ASSEMBLER INVOKED BY: A251.EXE DEMO.A51

LOC    OBJ             LINE     SOURCE

                          1     $TITLE (ASSEMBLER DEMO PROGRAM)
                          2     ; A simple Assembler Module for Demonstration
                          3
                          4     ; Symbol Definition
 00000D                   5     CR      EQU  13         ; Carriage-Return
 00000A                   6     LF      EQU  10         ; Line-Feed
                          7
                          8     ; Segment Definition
------                    9     ?PR?DEMO  SEGMENT  CODE ; Program Part
------                   10     ?CO?DEMO  SEGMENT  CODE ; Constant Part
                         11
                         12     ; Extern Definition
                         13     EXTRN CODE (PRINTS, DEMO)
                         14
                         15     ; The Program Start
000000                   16             CSEG  AT  0     ; Reset Vector
000000 020000     F      17             JMP   Start
                         18
------                   19             RSEG  ?PR?DEMO  ; Program Part
000000 900000     F      20     START:  MOV   DPTR,#Txt ; Demo Text
000003 120000     E      21             CALL  PRINTS    ; Print String
                         22     ;
000006 020000     E      23             JMP   DEMO      ; Demo Program
                         24
                         25     ; The Text Constants
------                   26             RSEG  ?CO?DEMO  ; Constant Part
000000 48656C6C          27     Txt:    DB    'Hello World',CR,LF,0
000004 6F20576F
000008 726C640D
00000C 0A00
                         28
                         29             END             ; End of Module

SYMBOL TABLE LISTING
------ ----- -------

N A M E             T Y P E  V A L U E   ATTRIBUTES

?CO?DEMO . . . . .  C  SEG   00000EH     REL=UNIT, ALN=BYTE
?PR?DEMO . . . . .  C  SEG   000009H     REL=UNIT, ALN=BYTE
CR . . . . . . . .  N  NUMB  00000DH A
DEMO . . . . . . .  C  ADDR  -------     EXT
LF . . . . . . . .  N  NUMB  00000AH A
PRINTS . . . . . .  C  ADDR  -------     EXT
START. . . . . . .  C  ADDR  000000H R   SEG=?PR?DEMO
TXT. . . . . . . .  C  ADDR  000000H R   SEG=?CO?DEMO

REGISTER BANK(S) USED: 0
ASSEMBLY COMPLETE.  0 WARNING(S), 0 ERROR(S)
```

† New features in the A251 assembler and the MCS 251 architecture

To assemble this module, the assembler was invoked using the following command line:

```
A251 DEMO.A51
```

The assembler output for this command line is:

```
DOS MACRO ASSEMBLER A251 V1.00


ASSEMBLY COMPLETE, NO ERRORS FOUND
```

After assembly, the object modules are linked and all variables and addresses are resolved and located into an executable program by the L251 linker. The linker is invoked with the following command line.

```
L251 DEMO.OBJ
```

The linker generates an absolute object file as well as a listing file and screen messages. The screen output for the linker is:

```
DOS LINKER/LOCATER L251 V1.00

L251 LINKING COMPLETE,  0 WARNINGS,  0 ERRORS
```

Chapter 2. 8051 and MCS 251 Architecture

This part reviews the existing 8051 memory and register architecture, before we introduce the MCS 251 architecture. Also described will be the salient differences between the 8051 microcontroller and the MCS 251 architecture. This part will only touch upon the hardware issues.

The A251 macro assembler is capable of generating code for both processor families with equal ease. The A251 assembler can be called upon to translate code written for the 8051 family of microcontrollers, generate native code for the 8051, or native code directly for the MCS 251 microcontroller.

In the following both processor architectures are explained.

New Features of the MCS 251 Architecture

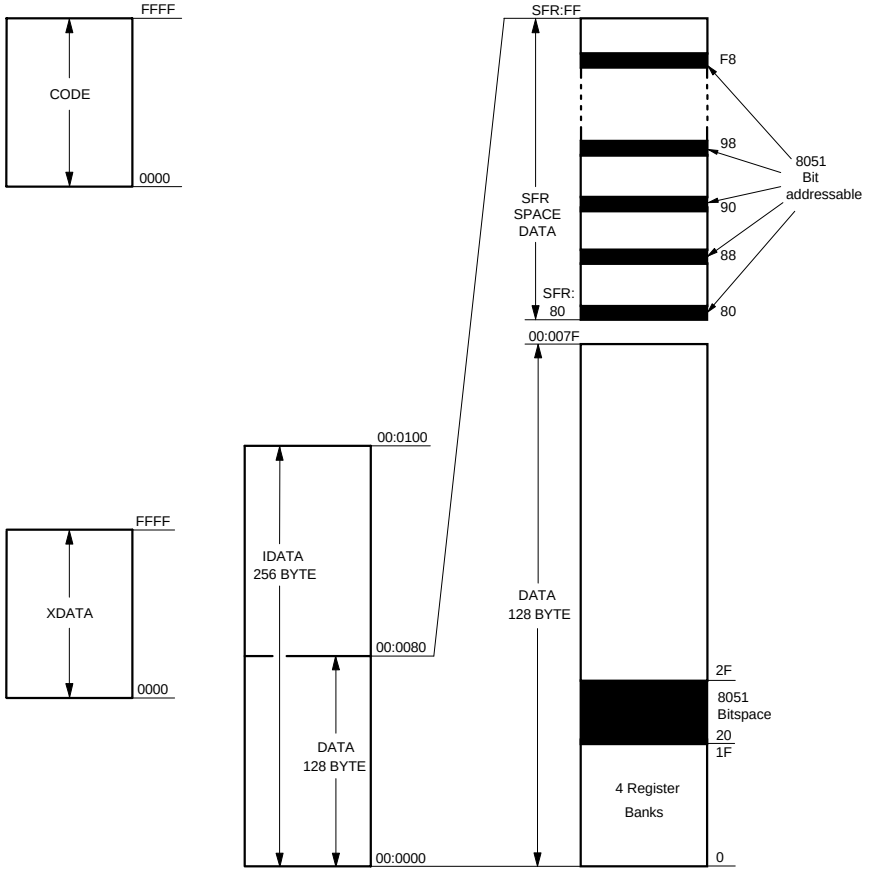
The MCS 251 instruction set is a superset of the standard MCS 51 microcontroller. The basic MCS 251 features:

- Completely code compatible with the MCS 51 microcontroller.
- Powerful 8/16/32-bit instructions.
- Flexible 8/16/32-bit register.
- 16MB linear address space; can be accessed fully by existing 8051 software; external code-banking logic is not required!
- The 251 can run your 51 programs up to 5 times faster.
- C applications re-translated with the C251 compiler are up to 15 times faster.
- True stack-oriented instruction set with 16-bit stack pointer.
- Direct CPU support for 16-bit and 32-bit pointers.

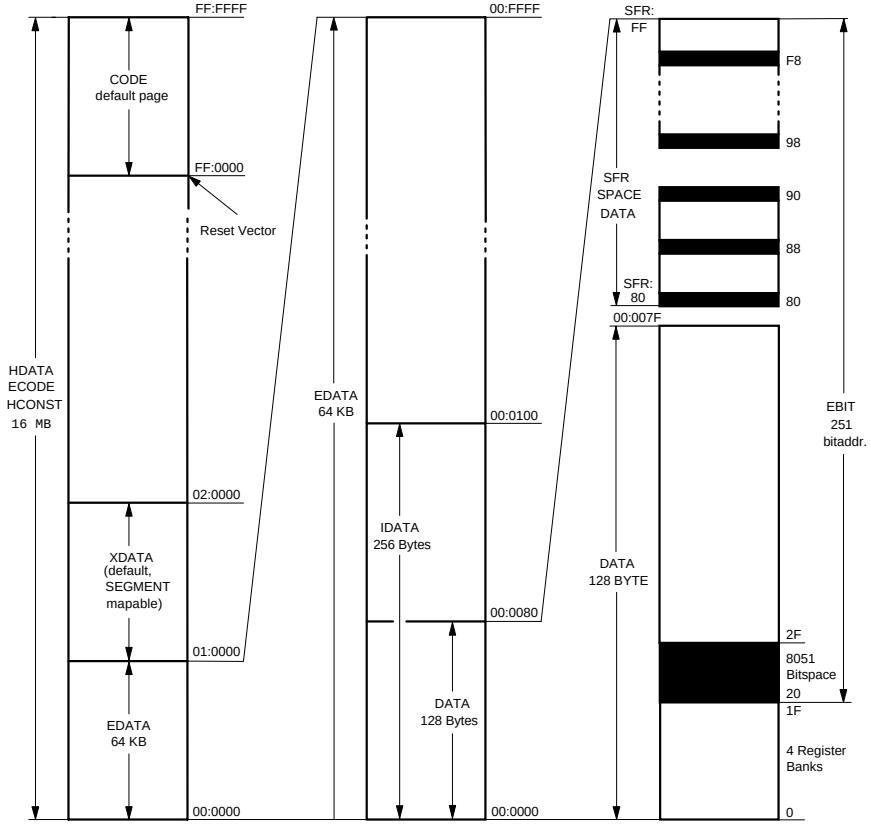
2

8051 and MCS 251 Memory Model

The standard 8051 memory model, shown in the following figure, is familiar to 8051 users the world over.



The following figure shows the memory model of the MCS 251 architecture.



The MCS 251 controller completely supports all aspects of the standard 8051 instruction set and memory organization. This ensures that all existing 8051 programs will successfully execute on the MCS 251. The 8051 family architecture has 4 separate address spaces: Program memory, Special Function registers, Internal and External Data memory.

8051 Address Space

All four 8051 memory spaces (DATA, IDATA, CODE and XDATA) are fully supported by the MCS 251 architecture by mapping them into separate regions in the MCS 251 address space. The four address spaces are integrated into one address space, yet they retain their 8051 microcontroller identity guaranteeing run-time compatibility with the 8051 microcontroller. The mapping is

completely transparent to the user and is taken care of by the A251 assembler and L251 linker.

Program Memory

The 8051 microcontroller Program Memory space is mapped at FF0000H, which is the MCS 251 “RESET” vector. All 8051 microcontroller instructions will work just as before in the 64K region starting at FF0000H. The MOVC instruction accesses the current active 64K segment, providing 8051 microcontroller compatibility. The A251 assembler translates 8051 microcontroller code in this 64K region making the mapping transparent to the user. All ORG statements are interpreted with this mapping. The reset and interrupt vectors are corresponding mapped, avoiding any problems on reset or interrupts.

Internal Data Memory

The internal data memory is mapped to location 0 ensuring complete run-time compatibility. Register banking, bit addressing, direct/indirect addressing as well as stack access are compatible to the 8051 microcontroller. The MCS 251 address space begins as 8051 microcontroller internal data memory and extends to 16M. This allows enhanced data/stack access using new instructions while maintaining compatibility with the existing 8051 microcontroller family.

External Memory

The 64K 8051 microcontroller external data memory is mapable to any segment within the 64KB memory space. After Reset the XDATA space is mapped to the area 64KB .. 128KB. This provides complete run-time compatibility with the 8051 microcontroller, since the lowest 16 address bits of the external data memory are identical to the standard 8051 controller. Keeping internal and external data memory spaces separated ensures that MOVX instructions does not access internal memory, and that 8051 microcontroller MOV instructions will not access external memory.

Memory Classes

Several new memory groups have been defined to take advantage of the 251 extended code and data capability. For convenience we refer to these as, Memory Classes. Each class has specific requirements and capabilities. These differences are listed below.

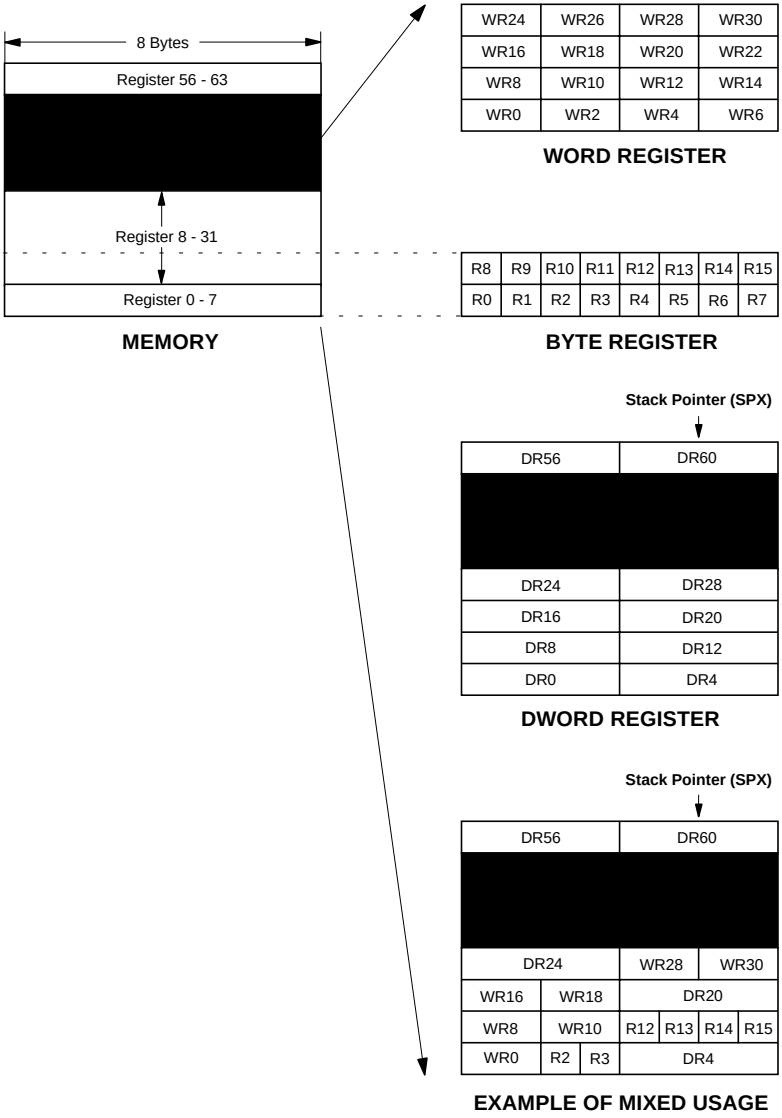
Memory Class	Address Range	Description
DATA	00:0000 - 00:007F	Direct addressable on-chip RAM.
BIT	00:0020 - 00:002F	8051 compatible bit-addressable RAM; can be accessed with short 8-bit addresses.
IDATA	00:0000 - 00:00FF	Indirect addressable on-chip RAM; can be accessed with @R0 or @R1.
EDATA	00:0000 - 00:FFFF	Extended direct addressable memory area; can be accessed with direct 16-bit addresses available on the 251.
ECONST	00:0000 - 00:FFFF	Same as EDATA - but allows the definition of ROM constants.
EBIT	00:0020 - 00:007F	Extended bit-addressable RAM; can be accessed with the extended bit addressing mode available on the 251.
XDATA	01:0000 - 01:FFFF (default space)	8051 compatible DATA space. Can be mapped on the 251 to any 64 KB memory segment. Accessed with MOVX instruction.
HDATA	00:0000 - FF:FFFF	Full 16 MB address space of the 251. Accessed with MOV @DRK instructions. This space is used for RAM areas.
HCONST	00:0000 - FF:FFFF	Same as HDATA - but allows the definition of ROM constants.
ECODE	00:0000 - FF:FFFF	Full 16 MB address space of the 251; executable code accessed with ECALL or EJMP instructions.
CODE	FF:0000 - FF:FFFF (default space)	8051 compatible CODE space; used for executable code or RAM constants. Can be located with L251 to any 64 KB segment
CONST	FF:0000 - FF:FFFF (default space)	Same as CODE - but can be used for ROM constants only.

8051 and MCS 251 Register File

The MCS 251 architecture supports an extra 32 bytes of register in addition to the 4 banks of 8 registers that the 8051 microcontroller architecture. The lower 8 byte registers are mapped between location 00:00 - 00:01FH. The lower 8 byte registers are mapped in this way to support 8051 microcontroller register banking (see the following figure). The register-file can be addressed in the following ways, depending upon the register accessed:

- Register 0 - 15 can be addressed as either byte, word, or double word (Dword) registers.
- Register 16 - 31 can be addressed as either word or Dword registers.
- Register 0 - 15 can be addressed only as Dword registers.
- There are 16 possible byte registers (R0 - R15), 16 possible word registers (WR0 - WR30) and 10 possible Dword registers (DR0 - DR28, DR56 - DR60) that can be addressed in any combination.
- All Dword registers are Dword aligned; each is addressed as Drk with “k” being the lowest of the 4 consecutive registers. For example, DR4 consists of registers 4 - 7.
- All word registers are word aligned; each is addressed as Wrj with “j” being the lower of the 2 consecutive registers. For example WR4 consists of registers 4 - 5.
- All byte registers are inherently byte aligned; each is addressed as Rm with “m” being the register number. For example R4 consists of register 4.

The following figure shows the register file format for the MCS 251 microcontroller.



Special Function Registers

The 128-byte SFR space is completely compatible with direct addressing of the 8051 controller SFRs including bit addressing. The address/data SFRs such as A, B, DPL, DPH, SP reside in the MCS 251 register file for high performance, however, they are also mapped into the 128-byte SFR region for compatibility. In the MCS 251 architecture, these SFRs can be referred to either by their 8051 microcontroller names, 8051 microcontroller addresses, or the new MCS 251 register names.

The following table shows how the MCS 51 microcontroller registers appear in the MCS 251 architecture.

MCS 51 microcontroller SFR Name	MCS 51 microcontroller SFR Address	Register in 251 Register file	251 Register Name
R0 to R7	-	0 through 7	**R0 to R7**
ACC	E0	11	**R11**
B	F0	10	**R10**
DPH	83	58	**DR56**
DPL	82	59	**DR56**
SP	81	63	**DR60 (SPX)**

For purpose of compatibility the Program Status Word (PSW) of the 8051 microcontroller has been left unchanged.

Differences to the 8051

The MCS 251 microcontroller uses the von Neumann Architecture for flexibility and simplicity. This means that code and data areas share a single contiguous memory address space.

The increased instruction throughput and instruction fetch rates of the 251 will require adjustments to code that is instruction cycle or timing dependent.

The extended memory and code space enables to work free of the 8051's historical restrictions.

8051 Compatibility

The A251 assembler will assemble existing 8051 microcontroller code for the MCS 251 without requiring any changes in the assembly code except for a few cases described below, where a assembler source needs changes under user control.

Timing Issues

The MCS 251 CPU significantly improves code performance; instructions are executed about 5 times faster than typical 8051 microcontroller execution. For example, the instruction ADD A,R*n* instruction takes 6 states on the 8051 microcontroller and 1 state on the MCS 251. Some instructions are executed up to 12 times faster.

Due to these intrinsic performance increases, special care must be given to the timing loops of 8051 microcontroller code assembled for the MCS 251. Additionally, 8051 microcontroller peripherals that rely on a time base may require adjustment before assembling.

MCS 251 timing issues encountered by existing 8051 microcontroller code would be the same as if the clock speed of the 8051 microcontroller were increased from 12MHz to 60MHz.

Stack Pointer (SPX)

In addition to being a word register, DR60 is also the 16-bit stack pointer. It is used for all the stack operations such as pushes/pops, call/returns, transfer to interrupt service routine and return from interrupt service routine. Making the stack pointer part of the register file allows all MCS 251 instructions to be used for stack pointer manipulation, and enhances stack access through a rich set of addressing modes.

Program Status Word

The Program Status Word (PSW) contains status bits that reflect the current state of the CPU. It consists of two 8-bit registers, PSW and PSW1. The PSW register retains the existing 8051 microcontroller flags and the PSW1 register

contains the new MCS 251 flags as well as the CY, AC, and OV. The Z flag will be set if the result of the last arithmetic or logical operation was a zero. The N flag will be set if the result of the last logical operation was negative.

PSW Bit Definitions

PSW Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
CY	AC	F0	RS1	RS0	OV	UD	P

PSW1 Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
CY	AC	N	RS1	RS0	OV	Z	–

The following table describes the bits in the PSW.

Symbol	Function
CY	Carry flag
AC	Auxiliary Carry flag (For BCD Operations)
F0	Flag 0 (Available to the user for General Purpose)
RS1, **RS0**	Register bank select bit 1 Register bank select bit 0 RS1 RS0 Working Register Bank and Address 0 0 Bank0 (00:00H - 00:07H) 0 1 Bank1 (00:08H - 00:0FH) 1 0 Bank2 (00:10H - 00:17H) 1 1 Bank3 (00:18H - 00:1FH)
OV	Overflow flag
UD	User definable flag
P	Parity flag
–	Reserved for future use
Z	Zero flag
N	Negative flag

Chapter 3. Writing Assembly Programs

The A251 macro assembler is a two pass assembler that translates 8051 assembly language programs into Intel compatible object files. These object files are then combined or linked using the Linker/Locator to form an executable, ready to run, absolute object module. As a subsequent step, absolute object modules can be converted to Intel HEX files suitable for loading onto to your target hardware, device programmer, or ICE (In-Circuit Emulator) unit.

The following sections describes the components of an assembly program, and some aspects of writing assembly programs. An assembly program consists of one or more statements. These statements contain directives, controls, and instructions.

Assembly Statements

Assembly program source files are made up of statements which may include assembler controls, assembler directives, or 8051 assembly language instructions (mnemonics). For example:

```
$TITLE(Demo Program #1)
                    ORG     0000h
                    JMP     $
                    END
```

This example program consists of four statements. **$TITLE** is an assembler control, **ORG** and **END** are assembler directives, and **JMP** is an assembly language instruction.

Each line of an assembly program can contain only one control, directive, or instruction statement. Statements must be contained in exactly one line. Multi–line statements are not allowed.

Statements in 8051/251 assembly programs are not column sensitive. Controls, directives, and instructions may start in any column. Indentation used in the examples in this manual, is done for program clarity and is neither required nor expected by the assembler. The only exception is that arguments and instruction operands must be separated from controls, directives, and instructions by at least one space.

All 8051/251 assembly programs must include the **END** directive. This directive signals to the assembler that this is the end of the assembly program. Any instructions, directives, or controls found after the **END** directive are ignored. The shortest valid assembly program contains only an **END** directive.

Directives

Assembler directives provide the assembly programmer with a means to instruct the assembler how to process subsequent assembly language instructions. Directives also provide a way for you to define program constants and reserve space for dynamic variables.

"Chapter 4. Assembler Directives" on page 41 provides complete descriptions and examples of all of the assembler directives that you may include in your program. Refer to this chapter for more information about how to use directives.

Controls

Assembler controls direct the operations of the assembler when generating a listing file or object file. Typically, controls do not impact the code that is generated by the assembler. Controls can be specified on the command line or within an assembler source file.

The conditional assembly controls are the only assembler controls that will impact the code that is assembled by the A251 assembler. The **IF**, **ELSE**, **ENDIF**, and **ELSEIF** controls provide a powerful set of conditional operators that can be used to include or exclude certain parts of your program from the assembly.

"Chapter 7. Invocation and Controls" on page 113 describes the available assembler controls in detail and provides an example of each. Refer to this chapter for more information about control statements.

Instructions

Assembly language instructions specify the program code that is to be assembled by the A251 assembler. The A251 assembler translates the assembly

instructions in your program into machine code and stores the resulting code in an object file.

Assembly instructions have the following general format:

```
[label:] mnemonic [operand] [, operand] [, operand] [; comment]
```

where

label	is a symbol name that is assigned the address at which the instruction is located.
mnemonic	is the ASCII text string that symbolically represents a machine language instruction.
operand	is an argument that is required by the specified ***mnemonic***.
comment	is an optional description or explanation of the instruction. A comment may contain any text you wish. Comments are ignored by the assembler.

The 8051 and 251 instructions are listed in “Appendix A. 8051/251 Instruction Sets” on page 173 by mnemonic and by machine language opcode. Refer to this section for more information about assembler instructions.

Comments

Comments are lines of text that you may include in your program to identify and explain the program. Comments are ignored by the A251 assembler and are not required in order to generate working programs.

You can include comments anywhere in your assembler program. Comments must be preceded with a semicolon character (**;**). A comment can appear on a line by itself or can appear at the end of an instruction. For example:

```
;This is a comment
                    NOP         ;This is also a comment
```

When the assembler recognizes the semicolon character on a line, it ignores subsequent text on that line. Anything that appears on a line to the right of a semicolon will be ignored by the A251 assembler. Comments have no impact on object file generation or the code contained therein.

Symbols

A symbol is a name that you define to represent a value, text block, address, or register name. You can also use symbols to represent numeric constants and expressions.

Symbol Names

Symbols are composed of up to 31 characters from the following list:

A - Z, a - z, 0 - 9, _, and **?**

A symbol name can start with any of these characters *except* the digits 0 - 9.

Symbols can be defined in a number of ways. You can define a symbol to represent (or EQUate to) an expression using the **EQU** or **SET** directives:

```
NUMBER_FIVE             EQU     5
TRUE_FLAG               SET     1
FALSE_FLAG              SET     0
```

you can define a symbol to be a label in your assembly program:

```
LABEL1:                 DJNZ    R0, LABEL1
```

and you can define a symbol to refer to a variable location:

```
SERIAL_BUFFER           DATA    99h
```

Symbols are used throughout an assembly program. Symbols provide better human understandable program element attributes. The following sections provide more information about the use and definition of symbols.

Labels

A label is a type of symbol that you define. A label defines a “place”. A labels name represents an address. All rules that apply to symbol names also apply to labels. When defined, a label must be the first text field in a line but may be preceded by tabs or spaces. A colon character (**:**) must immediately follow the symbol name to identify it as a label. Only one label can be defined on a line. For example:

```
LABEL1:         DS      2
LABEL2:                                         ;label by itself
NUMBER:         DB      27, 33, 'STRING', 0     ;label at a message
COPY:           MOV     R6, #12H                ;label in a program
```

In the above examples, **LABEL1, LABEL2, NUMBER**, and **COPY** are all labels.

When a label is defined, it receives the current value of the location counter of the currently selected segment. Refer to “Location Counter” on page 32 for more information about the location counter.

You can use a label just like you would use a program offset within an instruction. Labels can refer to program code, to variable space in internal or external data memory, or can refer to constant data stored in the program or code space.

You can use a label to transfer program execution to a different location. The instruction immediately following a label can be referenced by using the label. Your program can jump to or make a call to the label. The code immediately following the label will be executed.

You can also use labels to provide information to simulators and debuggers. A simulator or debugger can provide the label symbols while debugging. This can help to simplify the debugging process.

Labels may only be defined once. They may not be redefined.

Operands

Operands are arguments, or expressions, that are specified along with assembler directives or instructions. Assembler directives require operands that are constants or symbols. For example:

```
VVV             EQU     3
                DS      10h
```

Assembler instructions support a wider variety of operands than do directives. Some instructions require no operands and some may require up to 3 operands. Multiple operands are separated by commas. For example:

```
MOV             R2, #0
```

The number of operands that are required and their types depend on the instruction or directive that is specified. In the following table the first four operands can also be expressions. Instruction operands can be classified as one the following types:

Operand Type	Description
Immediate Data	Symbols or constants the are used as an numeric value.
Direct Bit Address	Symbols or constants that reference a bit address.
Program Addresses	Symbols or constants that reference a code address.
Direct Data Addresses	Symbols or constants that reference a data address.
Indirect Addresses	Indirect reference to a memory location, optionally with offset.
Special Assembler Symbol	Register names.

Special Assembler Symbols

The A251 assembler defines and reserves names of the 8051 register set. These predefined names are used in 8051 programs to access the 8051 processor registers.

Following, is a list of the each of the 8051 registers along with a brief description:

Register	Description
A	Represents the 8051 Accumulator. It is used with many operations including multiplication and division, moving data to and from external memory, boolean operations, etc.
DPTR	The DPTR register is a 16-bit data pointer used to address data in XDATA or CODE memory.
PC	The PC register is the 16-bit program counter. It contains the address of the next instruction to be executed.
C	The Carry flag; indicates the status of operations that generate a carry bit. It is also used by operations that require a borrow bit.
AB	The A and B register pair used in MUL and DIV instructions.
R0 – R7	The eight 8-bit general purpose 8051 registers in the currently active register bank. A Maximum of four register banks are available.
AR0 – AR7	Represent the absolute data addresses of R0 through R7 in the current register bank. The absolute address for these registers will change depending on the register bank that is currently selected. These symbols are only available when the USING directive is given. Refer to the USING directive for more information on selecting the register bank. These representations are suppressed by the NOAREGS directive. Refer to the NOAREGS directive for more information.
R8 - R15 †	Additional eight 8–bit general purpose registers of the 251.
WR0 - WR30 †	Sixteen 16–bit general purpose registers of the 251. The registers WR0 - WR14 overlap the registers R0 - R15. Note that there is no WR1 available.
DR0 - DR28 † **DR56** † **DR60** †	Ten 32-bit general purpose registers of 251. The registers DR0 - DR28 overlap the registers WR0 - WR30. Note that there is no DR1, DR2 and DR3 available.

Immediate Data

An immediate data operand is a numeric expression that is encoded as a part of the machine language instruction. Immediate data values are used literally in an instruction to change the contents of a register or memory location. The pound (or number) sign **(#)** must precede any expression that is to be used as an immediate data operand. The following shows some examples of how the immediate data is typically used:

```
MOV             A, #0E0h            ; load 0E0h into the accumulator
MOV             DPTR, #8000h        ; load 8000h into the data pointer
ANL             A, #128             ; AND the accumulator with 128
XRL             R0, #0FFh           ; XOR R0 with 0ffh
MOV             R5, #BUFFER         ; load R5 with the value of BUFFER
```

† New features in the A251 assembler and the MCS 251 architecture

Indirect Addresses

With indirect address operands it is possible to access the following memory classes of the 8051/251:

IDATA

Elements of this type must be accessed via registers R0 or R1. If these data elements also exist within the DATA memory class, 0H .. 07FH, then you may also access them directly.

Example

```
; BUFFER is a symbol with class IDATA or DATA.

MOV          R0,#BUFFER       ; load the address
MOV          A,@R0            ; the indirect access
```

XDATA

XDATA memory can be accessed with the instruction MOVX via the register DPTR or via the registers R0, R1.

Example

```
; XBUFFER is a symbol with class XDATA.
MOV               DPTR,#XBUFFER        ; load address
MOVX              @DPTR,A              ; access via DPTR
MOV               R1,#XBUFFER          ; load address
MOVX              A,@R1                ; access via R0 or R1
```

CODE and CONST †

CODE or CONST memory can be accessed with the instruction MOVC via the DPTR register.

Example

```
; TABLE is a symbol of class CODE or NCONST
MOV               DPTR,#TABLE        ; load address of table
MOV               A,#3               ; load offset into table
MOVC              A,@A+DPTR          ; access via MOVC instruction
```

† New features in the A251 assembler and the MCS 251 architecture

EDATA †

EDATA memory can be accessed via the registers WR0 .. WR30. Also variables of the class IDATA and DATA can be access with this addressing mode.

Example

```
; STRING is a symbol of class NDATA
MOV                WR4,#STRING          ; load address of STRING
MOV                R6,@WR4              ; indirect access
MOV                @WR4+2,R6            ; access with constant offset
```

HDATA †

HDATA memory can be accessed via the registers DR0 .. DR28. Any memory location can be accessed with these instructions.

Example

```
; ARRAY is a symbol of class HDATA
MOV               WR8,#WORD2 ARRAY      ; load address of ARRAY
MOV               WR10,#WORD0 ARRAY     ; into DR8
MOV               R4,@DR8               ; indirect access
MOV               @DR8+50H,R4           ; access with constant offset
```

Direct Data Addresses

Direct Data addresses represent the exact address of the data to access in the memory. Also the special function registers of the 8051 can be accessed with direct data addresses. With direct data address operands you can access the following memory classes of the 8051/251:

```
; accesses to DATA space
VALUE              DATA      20H
                   MOV       50H,A
                   MOV       R0,VALUE

; accesses to EDATA space
EVAR               EDATA     1000H
                   MOV       R5,EDATA 2000H
                   MOV       EVAR,R4
```

† New features in the A251 assembler and the MCS 251 architecture

Direct Bit Addresses

Direct bit addresses represent the exact address of the bit to access in the memory. Also the special function registers of the 8051/251 are bit addressable and can be accessed with direct bit addresses.

Bit addresses can be accessed using the period (**.**) to access the bits of byte variables that reside in the bit–addressable area (20h to 2Fh) or to access the bits of certain special function registers. The period must be specified after a byte base symbol and must have a trailing bit position to access.

With direct bit address operands you can access the following memory classes of the 8051/251:

```
; accesses to BIT class
; also variables with the class DATA BITADDRESSABLE can be accessed
               SETB      20H.6     ; set bit 6 in location 20H
               CLR       10        ; clear bit 2 in location 21H, this is
                                   ; the bit address 10
               MOV       C,ACC.5   ; move bit 5 of register A to the
                                   ; carry flag.

; accesses to EBIT space
; also variables with the class DATA can be accessed.
               MOV       40H.5,C
               SETB      DPL.7     ; set bit 7 in the register DPL
```

Program Addresses

Program addresses are absolute or relocatable expressions with the memory class CODE or ECODE. There are four types of instructions that require a program address in their operands:

Relative Jumps

Relative jumps include conditional jumps (**CJNE**, **DJNZ**, **JB**, **JBC**, **JC**, …) and the unconditional **SJMP** instruction. The addressable offset is –128 to +127 bytes from the first byte of the instruction that follows the relative jump. When you use a relative jump in your code, you must use an expression that evaluates to the code address of the jump destination. The assembler does all the offset computations. If the address is out of range, the assembler will issue an error message.

In-Block Jumps and Calls (ACALL and AJMP)

In-block jumps and calls permit access only within a 2KByte block of program space. The low order 11 bits of the program counter are replaced when the jump or call is executed.

If ACALL or AJMP is the last instruction in a block, the high order bits of the program counter change when incremented to address the next instruction.; thus the jump will be made within the block following the ACALL or AJMP.

Long Jumps and Calls (LJMP and LCALL)

Long jumps and calls allow to access within a 64KByte segment of program space. The low order 16 bits of the program counter are replaced when the jump or call is executed.

For the 8051 only: **LJMP** and **LCALL** can access the entire 8051 address space.

For the 251 only: If **LJMP** and **LCALL** is the last instruction in a segment, the high order bits of the program counter change when incremented to address the next instruction; thus the jump will be made within the block following the **LJMP** or **LCALL**.

Extended Jumps and Calls (EJMP and ECALL)

Extended jumps and calls allow access within the 16MByte program space of the 251. The low order 24 bits of the program counter are replaced when the jump or call is executed.

Generic Jump and Call (JMP and CALL)

The assembler provides two instruction mnemonics that do not represent a specific opcode. The are **JMP** and **CALL**. **JMP** may assemble to **SJMP**, **AJMP**, **LJMP** or **EJMP**. **CALL** may assemble to **ACALL**, **LCALL** or **ECALL**. These generic mnemonics will always evaluate to an instruction, not necessarily the shortest, that will reach the specified program address operand.

This is an effective tool to use during program development, since sections of code change drastically in size with each development cycle. Note that the assembler decision may not be optimal. For example, if the code address is a forward reference, the assembler will generate a long jump although a short jump may be possible.

Expressions and Operators

An operand may be a numeric constant, a symbolic name, a character string or an expression.

Operators are used to combine and compare operands within your assembly program. Operators are not assembly language instructions nor do they generate 8051 assembly code. They represent operations that are evaluated at assembly-time. Therefore, operators can only handle calculations of values that are known when the program is assembled.

An expression is a combination of numbers, character string, symbols, and operators that evaluate to a single 32-bit binary number. Expressions are evaluated at assembly time and can, therefore, be used to calculate values that would otherwise be difficult to determine beforehand.

The following sections describe operators and expressions and how they are used in 8051 assembly programs.

Numbers

Numbers can be specified in hexadecimal (base 16), decimal (base 10), octal (base 8), and binary (base 2). The base of a number is specified by the last character in the number. A number that is specified without an explicit base is interpreted as decimal number.

The following table lists the base types, the base suffix character, and some examples:

Base	Suffix	Legal Characters	Examples
Hexadecimal	H, h	0 – 9, A – F, a – f	1234h 99h 0A0F0h 0FFh
Decimal	D, d	0 – 9	1234 65590d 20d 123
Octal	O, o, Q, q	0 – 7	177o 25q 123o 177777q
Binary	B, b	0 and 1	10011111b 101010101b

The first character of a number must be a digit between 0 and 9. Hexadecimal numbers which do not have a digit as the first character should be prefixed with a 0.

The A251 assembler supports also hex numbers written in C notation. For example:

```
0xA0F0       0x24    0xff
```

The dollar sign character ($) can be used in a number to make it more readable, however, the dollar sign character cannot be the first or last character in the number. A dollar sign used within a number is ignored by the assembler and has no impact on the value of the number. For example:

```
1111$0000$1010$0011b          is equivalent to          1111000010100011B
1$2$3$4                       is equivalent to                       1234
```

Characters

The A251 assembler allows you to use ASCII characters in an expression to generate a numeric value. Up to two characters enclosed within single quotes (') may be included in an expression. More than two characters in single quotes in an expression will cause the A251 assembler to generate an error. Following are examples of character expressions:

```
'A'              evaluates to 0041h
'AB'             evaluates to 4142h
'a'              evaluates to 0061h
'ab'             evaluates to 6162h
''               null string evaluates to 0000h
'abc'            generates an ERROR
```

Characters may be used anywhere in your program as a immediate data operand. For example:

```
LETTER_A          EQU     'A'

TEST:             MOV     @R0, #'F'
                  SUBB    A, #'0'
```

Character Strings

Character strings can be used in combination with the **DB** directive to define messages that are used in your 8051 assembly program. Character strings must be enclosed within single quotes ('). For example:

```
KEYMSG:           DB      'Press any key to continue.'
```

generates the hexadecimal data (50h, 72h, 65h, 73h, 73h, 20h, … 6Eh, 75h, 65h, 2Eh) starting at **KEYMSG**. You can mix string and numeric data on the same line. For example:

```
EOLMSG:           DB      'End of line', 00h
```

appends the value 00h to the end of the string **'End of line'**.

Two successive single quote characters can be used to insert a single quote into a string. For example:

```
MSGTXT:           DB      'ISN''T A QUOTE REQUIRED HERE?'.
```

Location Counter

The A251 assembler maintains a location counter for each segment. The location counter contains the offset of the instruction or data being assembled and is incremented after each line by the number of bytes of data or code in that line.

The location counter is initialized to 0 for each segment, but can be changed using the **ORG** directive.

The dollar sign character (**$**) returns the current value of the location counter. This operator allows you to use the location counter in an expression. For example, the following code uses **$** to calculate the length of a message string.

```
MSG:              DB      'This is a message', 0
MSGLEN            EQU     $ - MSG
```

You can also use **$** in an instruction. For example, the following line of code will repeat forever.

```
JMP               $       ; repeat forever
```


Operators

The A251 assembler provides several classes of operators that allow you to compare and combine operands and expressions. These operators are described in the sections that follow.

Arithmetic Operators

Arithmetic operators perform arithmetic functions like addition, subtraction, multiplication, and division. These operators require one or two operands depending on the operation. The result is always a 16-bit value. Overflow and underflow conditions are not detected. Division by zero is detected and causes an assembler error.

The following table lists the arithmetic operators and provides a brief description of each.

Operator	Syntax	Description
+	+ *expression*	Unary plus sign
–	– *expression*	Unary minus sign
+	*expression* + *expression*	Addition
–	*expression* – *expression*	Subtraction
*****	*expression* * *expression*	Multiplication
/	*expression* / *expression*	Integer division
MOD	*expression* MOD *expression*	Remainder
(and)	(*expression*)	Specify order of execution

Binary Operators

Binary operators are used to complement, shift, and perform bit–wise operations on the binary value of their operands. The following table lists the binary operators and provides a brief description of each.

Operator	Syntax	Description
NOT	NOT *expression*	Bit–wise complement
SHR	*expression* SHR *count*	Shift right
SHL	*expression* SHL *count*	Shift left
AND	*expression* AND *expression*	Bit–wise AND
OR	*expression* OR *expression*	Bit–wise OR
XOR	*expression* XOR *expression*	Bit–wise exclusive OR

Relational Operators

The relational operators compare two operands. The results of the comparison is a TRUE or FALSE result. A FALSE result has a value of 0000h. A TRUE result has a non–zero value.

The following table lists the relational operators and provides a brief description of each.

Operator	Syntax	Result
GTE	*expression1* GTE *expression2*	True if *expression1* is greater than or equal to *expression2*
LTE	*expression1* LTE *expression2*	True if *expression1* is less than or equal to *expression2*
NE	*expression1* NE *expression2*	True if *expression1* is not equal to *expression2*
EQ	*expression1* EQ *expression2*	True if *expression1* is equal to *expression2*
LT	*expression1* LT *expression2*	True if *expression1* is less than *expression2*
GT	*expression1* GT *expression2*	True if *expression1* is greater than *expression2*
>=	*expression1* >= *expression2*	True if *expression1* is greater than or equal to *expression2*
<=	*expression1* <= *expression2*	True if *expression1* is less than or equal to *expression2*
<>	*expression1* <> *expression2*	True if *expression1* is not equal to *expression2*
=	*expression1* = *expression2*	True if *expression1* is equal to *expression2*
<	*expression1* < *expression2*	True if *expression1* is less than *expression2*

Operator	Syntax	Result
>	*expression1* > *expression2*	True if *expression1* is greater than *expression2*

Class Operators

The class operator assigns a memory class to an expression. This is how you associate an expression with a class. The A251 assembler generates an error message if you use an expression with a class on an instruction which does not support this class, for example, when you use an **HDATA** expression as a direct address.

The following table lists the class operators and provides a brief description of each.

Operator	Syntax	Description
BIT	BIT *expression*	Assigns the class BIT to the expression.
CODE	CODE *expression*	Assigns the class CODE to the expression.
CONST †	CONST expression	Assigns the class CONST to the expression.
DATA	DATA *expression*	Assigns the class DATA to the expression.
EBIT †	EBIT *expression*	Assigns the class EBIT to the expression.
ECODE †	ECODE *expression*	Assigns the class ECODE to the expression.
ECONST †	ECONST *expression*	Assigns the class ECONST to the expression.
EDATA †	EDATA *expression*	Assigns the class EDATA to the expression.
HCONST †	HCONST *expression*	Assigns the class HCONST to the expression.
HDATA †	HDATA *expression*	Assigns the class HDATA to the expression.
IDATA	IDATA *expression*	Assigns the class IDATA to the expression.
XDATA	XDATA *expression*	Assigns the class XDATA to the expression.

Type Operators †

The type operator assigns a data type to an expression. Thus the expression is associated with this type. A251 will generate an error message if you are using an expression with an type on an instruction which does not support this types. For example when you are using a WORD expression as argument in a byte-wide instruction of the 251.

The following table lists the type operators and provides a brief description of each.

Operator	Syntax	Description
BYTE	BYTE *expression*	Assigns the type BYTE to the expression.
WORD	WORD *expression*	Assigns the class WORD to the expression.
DWORD	DWORD *expression*	Assigns the class DWORD to the expression.
NEAR	NEAR *expression*	Assigns the class NEAR to the expression.
FAR	FAR *expression*	Assigns the class FAR to the expression.

Miscellaneous Operators

A251 provides operators that do not fall into the previously listed categories. These operators are listed and described in the following table.

Operator	Syntax	Description
LOW	LOW expression	Low–order byte of expression
HIGH	HIGH expression	High–order byte of expression
BYTE0 †	BYTE0 expression	Byte 0 of expression. See table below. (identical with LOW).
BYTE1 †	BYTE1 expression	Byte 1 of expression. See table below. (identical with HIGH).
BYTE2 †	BYTE2 expression	Byte 2 of expression. See table below.
BYTE3 †	BYTE3 expression	Byte 3 of expression. See table below.
WORD0 †	WORD0 expression	Word 0 of expression. See table below.
WORD2 †	WORD2 expression	Word2 of expression. See table below.

The following table shows how the byte and word operators impact a 32-bit value.

MSB			LSB
BYTE3	BYTE2	BYTE1	BYTE0
WORD2		WORD0	
		HIGH	LOW

† New features in the A251 assembler and the MCS 251 architecture

Operator Precedence

All operators are evaluated in a certain, well–defined order. This order of evaluation is referred to as operator precedence. Operator precedence is required in order to determine which operators are evaluated first in an expression. The following table lists the operators in the order of evaluation. Operators at level 1 are evaluated first. If there is more than one operator on a given level, the leftmost operator is evaluated first followed by each subsequent operator on that level.

Level	Operators
1	( )
2	**NOT**, **HIGH**, **LOW**, **BYTE0**, **BYTE1**, **BYTE2**, **BYTE3**, **WORD0**, **WORD2**
3 †	**BIT**, **CODE**, **CONST**, **DATA**, **EBIT**, **EDATA**, **ECONST**, **ECODE**, **HCONST**, **HDATA**, **IDATA**, **XDATA**
4 †	**BYTE**, **WORD**, **DWORD**, **NEAR**, **FAR**
5	+ (unary), – (unary)
6	*, /, **MOD**
7	+, –
8	**SHR**, **SHL**
9	**AND**, **OR**, **XOR**
10	>=, <=, =, <>, <, >, **GTE**, **LTE**, **EQ**, **NE**, **LT**, **GT**

Expressions

An expression is a combination of operands and operators that must be calculated by the assembler. An operand with no operators is the simplest form of an expression. An expression can be used in most places where an operand is required.

Expressions have a number of attributes that are described in the following sections.

Expression Classes

Expressions are assigned classes based on the operands that are used. The following classes apply to expressions:

Expression Class	Description
N NUMB	A classless number.
C ADDR	A **CODE** address symbol.
D ADDR	A **DATA** address symbol.
I ADDR	An **IDATA** address symbol.
X ADDR	An **XDATA** address symbol.
B ADDR	A **BIT** address symbol.
CO ADDR †	A **CONST** address symbol.
EC ADDR †	An **ECONST** address symbol.
CE ADDR †	An **ECODE** address symbol.
ED ADDR †	An **EDATA** address symbol.
EB ADDR †	An **EBIT** address symbol.
HD ADDR †	An **HDATA** address symbol.
HC ADDR †	An **HCONST** address symbol.

Typically, expressions are assigned the class **NUMBER** because they are composed only of numeric operands. You can assign a class to an expression using a class operand. An address symbol value gets automatically the class from the segment where it is defined. When an value has a class, a few rules apply to how expressions are formed:

1. The result of a unary operation has the same class as its operand.

2. The result of all binary operations except + and – will be a **NUMBER** type.

3. If only one of the operands of an addition or subtraction operation has a class, the result will have that class. If both operands have a class, the result will be a **NUMBER**.

This means that a class value (i.e. an addresses symbol) plus or minus a number (or a number plus a class value) give a value with class.

† New features in the A251 assembler and the MCS 251 architecture

Examples

```
data_address - 10                                  gives a data_address value
10 + edata_address                               gives an edata_address value
(data_address - data_address)                       gives a classless number
code_address + (data_address - data_address)     gives a code_address value
```

Expressions that have a type of **NUMBER** can be used virtually anywhere. Expressions that have a class can only be used where a class of that type is valid.

Relocatable Expressions

Relocatable expressions are so named because they contain a reference to a relocatable or external symbol. These types of expressions can only be partially calculated by the assembler since the assembler does not know the final location of relocatable segments. The final calculations are performed by the linker.

A relocatable expression normally contains only a relocatable symbol, however, it may contain other operands and operators as well. A relocatable symbol can be modified by adding or subtracting a constant value.

Examples for valid relocatable expression

- relocatable_symbol + absolute_expression
- relocatable_symbol - absolute_expression
- absolute_expression + relocatable_symbol

There are two basic types of relocatable expressions: simple relocatable expressions and extended relocatable expressions.

Simple Relocatable Expressions

Simple relocatable expressions contain symbols that are defined in a relocatable segment. Segment and external symbols are not allowed in simple relocatable expressions.

Simple relocatable expression can be used in four contexts:

1. As an operand to the ORG directive.
2. As an operand to a symbol definition directive (i.e. EQU, SET)
3. As an operand to a data initialization directive (DB, DW or DD)
4. As an operand to a machine instruction

Examples for simple relocatable expressions

```
REL1 + ABS1 * 10
REL2 - ABS1
REL1 + (REL2 - REL3)      assuming REL2 and REL3 refer to the same segment.
```

Invalid form of simple relocatable expressions

```
(REL1 + ABS1) * 10                relocatable value may not be multiplied.
(EXT1 - ABS1)                     this is a general relocatable expression
REL1 + REL2                            you cannot add relocatable symbols.
```

Extended Relocatable Expressions

The extended relocatable expressions have generally the same rules that apply to simple relocatable expressions. Segment and external symbols are allowed in extended relocatable expressions. Extended relocatable expression can be used only in statements that generate code as operands; these are:

- As an operand to a data initialization directive (DB, DW or DD)
- As an operand to a machine instruction

Examples for extended relocatable expressions

```
REL1 + ABS1 * 10
EXT1 - ABS1
LOW (REL1 + ABS1)
WORD2 (SEG1)
```

Invalid form of simple relocatable expressions

```
(SEG1 + ABS1) * 10                relocatable value may not be multiplied.
(EXT1 - REL1)                  you can add/subtract only absolute quantities

LOW (REL1) + ABS1                              LOW may be applied only to the
                                                 final relocatable expression
```

Chapter 4. Assembler Directives

This part describes the assembler directives. It shows how to define symbols and how to control the placement of code and data in program memory.

Introduction

The A251 assembler has several directives that permit you to define symbol values, reserve and initialize storage, and control the placement of your code.

The directives should not be confused with instructions. They do not produce executable code, and with the exception of the DB, DW and DD directives, they have no direct effect on the contents of code memory. These directives change the state of the assembler, define user symbols, and add information to the object file.

The directives are divided into the following categories:

- **Segment Control**
  Generic Segments: **SEGMENT**, **RSEG**
  Absolute Segments: **CSEG**, **DSEG**, **BSEG**, **ISEG**, **XSEG**
- **Symbol Definition**
  Generic Symbols: **EQU**, **SET**
  Address Symbols: **BIT**,.**CODE**, **DATA**, **IDATA**, **XDATA**
  Text Replacement: **LIT** †
- **Memory Initialization**
  DB, **DW**, **DD** †
- **Memory Reservation**
  DBIT, **DS**, **DSB** †, **DSW** †, **DSD** †
- **Procedure Declaration** †
  PROC / **ENDP** †, **LABEL** †
- **Program Linkage**
  PUBLIC, **EXTRN** / **EXTERN** †, **NAME**
- **Address Control**
  ORG, **EVEN** †, **USING**

- **Others**
  END

The A251 assembler is a two-pass assembler. In the first pass, symbols values are determined, and in the second, forward references are resolved, and object code is produced. This structure imposes a restriction on the source program: expressions which define symbol values (refer to "Symbol Definition" on page 51) and expressions which control the location counter (refer to "ORG" on page 67, "DS" on page 59, and "DBIT" on page 58) may not have forward references.

Segment Controls

A segment is a block of code or data memory the assembler creates from code or data in an 8051 assembly source file. How you use segments in your source modules depends on the complexity of your application. Smaller applications need less memory and are typically less complex than large multi–module applications.

The 8051 is a architecture CPU with specific memory areas. You use segments to locate program code, constant data, and variables in these areas.

Location Counter

A251 maintains a location counter for each segment. The location counter is a pointer to the address space of the active segment and represents an offset for generic segments or the actual address for absolute segments. When a segment is first activated, the location counter is set to 0. The location counter is changed after each instruction by the length of the instruction. The memory initialization and reservation directives (i.e. DS, DB or DBIT) change the value of the location counter as memory is allocated by these directives. The ORG directive sets a new value for the location counter. If you change the active segment and later return to that segment, the location counter is restored to its previous value. Whenever the assembler encounters a label it assigns the current value of the location counter and the type of the current segment to that label.

The dollar sign ($) indicates the value of the location counter in the active segment. When you use the $ symbol, keep in mind that its value changes with each instruction, but only after that instruction has been completely evaluated. If

you use $ in an operand to an instruction or directive, it represents the address of the first byte of that instruction.

The following sections describe the different types of segments.

Generic Segments

Generic segments have a name and a class as well as other attributes. Generic segments with the same name but from different object modules are considered to be parts of the same segment and are called partial segments. These segments are combined at link time by the linker/locator.

Generic segments are created using the **SEGMENT** directive. You must specify the name of the segment, the segment class, and an optional relocation type and alignment type when you create a relocatable segment.

Example

```
MYPROG    SEGMENT    CODE
```

defines a segment named **MYPROG** with a memory class of **CODE**. This means that data in the **MYPROG** segment will be located in the code or program area of the 8051. Refer to “SEGMENT” on page 46 for more information on how to declare generic segments.

Once you have defined a relocatable segment name, you must select that segment using the **RSEG** directive. When **RSEG** is used to select a segment, that segment becomes the active segment that A251 uses for subsequent code and data until the segment is changed with **RSEG** or with an absolute segment directive.

Example

```
RSEG    MYPROG
```

will select the MYPROG segment that is defined above.

Typically, assembly routines are placed in generic segments. If you interface your assembly routines to C, all of your assembly routines must reside in separate generic segments and the segment names must follow the standards used by C51. Refer to the *C51 Compiler User’s Guide* or the *C251 Compiler User’s Guide* for more information on interfacing assembler programs to C.

Stack Segment

The 8051 and MCS 251 architecture uses a hardware stack to store return addresses for **CALL** instructions and also for temporary storage using the **PUSH** and **POP** instructions. An 8051 application that uses these instructions must setup the stack pointer to an area of memory that will not be used by other variables.

For the **8051** a stack segment must be defined and space must be reserved as follows.

```
STACK           SEGMENT     IDATA
                RSEG        STACK           ; select the stack segment
                DS          10h             ; reserve 16 bytes of space
```

Then, you must initialize the stack pointer early in your program.

```
CSEG            AT          0               ; RESET Vector
                JMP         STARTUP         ; Jump to startup code
STARTUP:                                    ; code executed at RESET
                MOV         SP,#STACK - 1   ; load Stack Pointer
```

For the MCS 215 a stack segment must be defined and space must be reserved as follows.

```
STACK           SEGMENT     EDATA
                RSEG        STACK           ; select the stack segment
                DS          10h             ; reserve 16 bytes of space
```

Then, you must initialize the stack pointer early in your program.

```
CSEG            AT        0                 ; RESET Vector
                JMP       STARTUP           ; Jump to startup code
STARTUP:                                    ; code executed at RESET
                MOV       DR60,#STACK - 1   ; load Stack Pointer
```

If you are interfacing assembly routines to C, you probably do not need to setup the stack. This is already done for you in the C startup code.

Absolute Segments

Absolute segments reside in a fixed memory location. Absolute segments are created using the **CSEG**, **DSEG**, **XSEG**, **ISEG**, and **BSEG** directives. These directives allow you to locate code and data or reserve memory space in a fixed location. You use absolute segments when you need to access a fixed memory

location or when you want to place program code or constant data at a fixed memory address. Refer to the **CSEG**, **DSEG**, **ISEG**, **XSEG**, **ISEG** directives for more information on how to declare absolute segments.

After reset, the 8051 begins program executing at CODE address 0. The 251 starts execution at address FF0000. Some type of program code must reside at this address. You can use an absolute segment to force program code into this address. The following example is used in the C51 startup routines to branch from the reset address to the beginning of the initialization code.

```
.
.
.
                    CSEG      AT 0
RESET_VEC:          LJMP      STARTUP
.
.
.
```

The program code that we place at address 0000h (for 251 at address FF0000h) with the **`CSEG AT 0`** directive performs a jump to the **`STARTUP`** label.

A251 supports absolute segment controls for compatibility to A51. A251 translates the CSEG, DSEG, XSEG, ISEG and BSEG directives to a generic segment directive.

Default Segment

By default, A251 assumes that the CODE segment is selected and initializes the location counter to 0000h (FF0000h) when it begins processing an assembly source module. This allows you to create programs without specifying any relocatable or absolute segment directives.

SEGMENT

The **SEGMENT** directive is used to declare a generic segment. A relocation type and a allocation type may be specified in the segment declaration. The **SEGMENT** directive is specified using the following format:

```
segment  SEGMENT  class    reloctype      alloctype
```

where

segment is the symbol name to assign to the segment. this symbol name is referred by the following RSEG directive. The segment symbol name can be used also in expressions to represent the base or start address of the combined segment as calculated by the Linker/Locator.

class is the memory class to use for the specified segment. The class specifies the memory space for the segment. See the table below for more information.

reloctype is the relocation type for the segment. This determines what relocation options may be performed by the Linker/Locator. Refer to the table below for more information.

alloctype is the allocation type for the segment. This determines what relocation options may be performed by the Linker/Locator. Refer to the table below for more information.

Class

The name of each segment within a module must be unique. However, the linker will combine segments having the same segment type. This applies to segments declared in other source modules as well.

class specifies the memory class space for the segment. The A251 differs between basic classes and user defined classes. The *class* is used by the linker/locator to access all the segments which belong to that class.

The basic classes are listed below:

Basic Class	Description
BIT	BIT space (address 20H .. 2FH).

Basic Class	Description
CODE	CODE space (default for 251 address 0FF0000H .. 0FFFFFFH).
CONST †	CONST space; same as CODE but for constant only; access via MOVC.
DATA	DATA space (address 0 to 7FH & SFR registers).
EBIT †	Extended 251 bit space (address 20H .. 7FH)
EDATA †	EDATA space (address 0 .. 0FFFFH).
ECONST †	ECONST space; same as EDATA but for constants; (address 0 .. 0FFFFH).
IDATA	IDATA space (address 0 to 0FFH).
ECODE †	Entire 251 address space for program code.
HCONST †	Entire 251 address space for constants; access via MOV @DRk.
HDATA †	Entire 251 address space for data; access via MOV @DRk.
XDATA	XDATA space (default for 251 address 10000H .. 1FFFFH); access via MOVX.

User-defined Class Names †

User-defined class names are composed for a basic class name plus an extension. With user-defined class names you can access the same address space as with the basic class name. The advantage is that you can reference with the user defined class name all segment names which that name and direct them at the linker/locator level to a specific physical address. User-defined class names must be enclosed in quotation marks (').

Examples

```
seg1    SEGMENT    'NDATA_FLASH'
seg2    SEGMENT    'HCONST_BITIMAGE'
seg3    SEGMENT    'DATA1'
```

Relocation Type

The optional relocation type defines the relocation operation that may be performed by the Linker/Locator. The following table lists the valid relocation types:

Relocation Type	Description
AT *address*	Specifies an absolute segment. The segment will be placed at the specified *address*.
BITADDRESSABLE	Specifies a segment which will be located within the bit addressable memory area (20H to 2FH in DATA space). BITADDRESSABLE is only allowed for segments with the class DATA that do not exceed 16 bytes in length.

† New features in the A251 assembler and the MCS 251 architecture

Relocation Type	Description
INBLOCK	Specifies a segment which must be contained in a 2048Byte block. This relocation type is only valid for segments with the class CODE.
INPAGE	Specifies a segment which must be contained in a 256Byte page.
OFFS *offset* †	Specifies an absolute segment. The segment will be placed at the start address of the memory class plus the specified *offset*. The advantage compare to the AT relocation type is, that the start address of the memory class can be defined at Linker/Locator level. Refer to the *MCS 251 Utilities User's Guide* for more information.
OVERLAYABLE	Specifies that the segment can share memory with other segments. Segments declared with this relocation type can be overlaid with other segments which are also declared with the OVERLAYABLE relocation type. When using this relocation type, the segment name must be declared according to the C251, C51 or PL/M-51 segment naming rules. Refer to the *C51 Compiler User's Guide* or the *C251 Compiler User's Guide* for more information.
INSEG †	Specifies a segment which must be contained in a 64KByte segment.

Allocation Type

The optional allocation type defines the allocation operation that may be performed by the Linker/Locator. The following table lists the valid allocation types:

Allocation Type	Description
BIT †	Specify bit alignment for the segment. This is the default for all segments with the class BIT.
BYTE †	Specify byte alignment for the segment. This is the default for all segments except of BIT.
WORD †	Specify word alignment for the segment.
DWORD †	Specify dword alignment for the segment.
PAGE	Specify a segment whose starting address must be on a 256Byte page boundary.
BLOCK †	Specify a segment whose starting address must be on a 2048Byte block boundary.
SEG †	Specify a segment whose starting address must be on a 64KByte segment boundary.

Examples for Segment Declarations

```
IDS     SEGMENT     IDATA
```

Defines a segment with the name IDS and the memory class IDATA.

```
MYSEG   SEGMENT   CODE   AT 0FF2000H
```

Defines a segment with the name MYSEG and the memory class CODE to be located at address 0FF2000H.

```
HDSEG    SEGMENT    HDATA INSEG DWORD
```

Defines a segment with the name HDSEG and the memory class HDATA. The segment is located within one 64KByte segment and is DWORD aligned.

```
XDSEG     SEGMENT    XDATA PAGE
```

Defines a segment with the name XDSEG and the memory class XDATA. The segment is PAGE aligned, this means it starts on a 256Byte page.

```
HCSEG     SEGMENT    HCONST SEG
```

Defines a segment with the name HCSEG with the memory class HCONST. The segment is SEGMENT aligned, this means it starts on a 64KByte segment.

RSEG

The **RSEG** directive selects a generic segment that was previously declared using the **SEGMENT** directive. The **RSEG** directive uses the following format:

```
RSEG  segment
```

where

segment is the name of a segment that was previously defined using the **SEGMENT** directive. Once selected, the specified segment remains active until a new segment is specified.

Example

```
.
.
.
MYPROG              SEGMENT    CODE       ; declare a segment

                    RSEG       MYPROG     ; select the segment
                    MOV        A, #0
                    MOV        P0, A
.
.
.
```

BSEG, CSEG, DSEG, ISEG, XSEG

The **BSEG**, **CSEG**, **DSEG**, **ISEG**, **XSEG** directive selects an absolute segment. This directives are using the following format:

```
BSEG   AT  address     defines an absolute BIT segment.
CSEG   AT  address     defines an absolute CODE segment.
DSEG   AT  address     defines an absolute DATA segment.
ISEG   AT  address     defines an absolute IDATA segment.
XSEG   AT  address     defines an absolute XDATA segment.
```

where

address is an optional absolute base address at which the segment begins. The ***address*** may not contain any forward references and must be an expression that can be evaluated to a valid address.

CSEG, **DSEG**, **ISEG**, **BSEG** and **XSEG** select an absolute segment within the code, internal data, indirect internal data, bit, or external data address spaces. If you choose to specify an absolute address (by including **AT *address***), the assembler terminates the last absolute segment, if any, of the specified segment type, and creates a new absolute segment starting at that address. If you do not specify an address, the last absolute segment of the specified type is continued. If no absolute segment of this type was selected and the absolute address is omitted, a new segment is created starting at location 0. You cannot use any forward references and the start address must be an absolute expression.

The A251 Macro Assembler supports the **BSEG**, **CSEG**, **DSEG**, **ISEG**, and **XSEG** directives for A51 compatibility. These directives are converted to standard segments as follows:

A51 Directive	Converted to A251 Segment Declaration
BSEG AT 20H.1	?BI?*modulename?n* SEGMENT OFFS 20H.1
CSEG AT 1234H	?CO?*modulename?n* SEGMENT OFFS 1234H
DSEG AT 40H	?DT?*modulename?n* SEGMENT OFFS 40H
ISEG AT 80H	?ID?*modulename?n* SEGMENT OFFS 80H
XSEG AT 5100H	?XD?*modulename?n* SEGMENT OFFS 5100H

where

modulname is the name of the current assembler module

n is a sequential number incremented for every absolute segment.

Examples

```
                        BSEG AT 30h         ; absolute bit segment @ 30h
DEC_FLAG:               DBIT     1          ; absolute bit
INC_FLAG:               DBIT     1
                        CSEG AT 100h        ; absolute code segment @ 100h
PARITY_TAB:             DB       00h        ; parity for 00h
                        DB       01h        ;             01h
                        DB       01h        ;             02h
                        DB       00h        ;             03h
.
.
.
                        DB       01h        ;             FEh
                        DB       00h        ;             FFh
                        DSEG AT 40h         ; absolute data segment @ 40h
TMP_A:                  DS       2          ; absolute data word
TMP_B:                  DS       4
                        ISEG AT 40h         ; abs indirect data seg @ 40h
TMP_IA:                 DS       2
TMP_IB:                 DS       4

                        XSEG AT 1000h       ; abs external data seg @ 1000h
OEMNAME:                DS       25         ; abs external data
PRDNAME:                DS       25
VERSION:                DS       25
```

Symbol Definition

The symbol definition directives allow you to create symbols that can be used to represent registers, numbers, and addresses.

Symbols defined by these directives may not have been previously defined and may not be redefined by any means. The **SET** directive is the only exception to this.

EQU, SET

The **EQU** and **SET** directive assigns a numeric value or register symbol to the specified symbol name. Symbols defined with **EQU** may not have been previously defined and may not be redefined by any means. The **SET** directive allows later redefinition of symbols. Statements involving the **EQU** or **SET** directive are formatted as follows:

```
symbol      EQU     expression
symbol      EQU     register
symbol      SET     expression
symbol      SET     register
```

where

symbol	is the name of the symbol to define. The expression or register specified in the **EQU** or **SET** directive will be substituted for each occurrence of ***symbol*** that is used in your assembly program.
expression	is a numeric expression which contains no forward references, or a simple relocatable expression.
register	is one of the following register names: A, R0, R1, R2, R3, R4, R5, R6, or R7.

Symbols defined with the **EQU** or **SET** directive may be used anywhere in operands, expressions, or addresses. Symbols that are defined as a register name can be used anywhere a register is allowed. A251 replaces each occurrence of the defined symbol in your assembly program with the specified numeric value or register symbol.

Symbols defined with the **EQU** directive may not be changed or redefined. You cannot use the **SET** directive if a symbol was previously defined with **EQU** and you cannot use the **EQU** directive if a symbol which was defined with SET.

Examples

```
LIMIT       EQU     1200
VALUE       EQU     LIMIT – 200 + 'A'
SERIAL      EQU     SBUF
ACCU        EQU     A
COUNT       EQU     R5
VALUE       SET     100
VALUE       SET     VALUE / 2
COUNTER     SET     R1
TEMP        SET     COUNTER
TEMP        SET     VALUE * VALUE
```

CODE, DATA, IDATA, XDATA

The **BIT**, **CODE**, **DATA**, **IDATA**, and **XDATA** directives assigns an address value to the specified symbol. Symbols defined with the **BIT**, **CODE**, **DATA**,

IDATA, and **XDATA** directives may not be changed or redefined. The format of theses directives is:

```
symbol     BIT      bit_address                  defines a BIT symbol
symbol     CODE     code_address                defines a CODE symbol
symbol     DATA     data_address                defines a DATA symbol
symbol     IDATA    idata_address              defines an IDATA symbol
symbol     XDATA    xdata_address               defines a XDATA symbol
```

where

symbol is the name of the symbol to define. The symbol name can be used anywhere an address of this memory class is valid.

bit_address is the address of a bit in internal data memory in the area 20H .. 2FH or a bit address of an 8051 bit-addressable SFR.

code_address is a code address in the range 0000H .. 0FFFFH.

data_address is a data memory address in the range 0 to 127 or a special function register (SFR) address in the range 128 .. 255.

idata_address is an idata memory address in the range 0 to 255.

xdata_address is an xdata memory address in the range 0 to 65535.

Example

```
DATA_SEG        SEGMENT BITADDRESSABLE
RSEG            DATA_SEG                    ; a bitaddressable rel_seg

CTRL:           DS      1                   ; a 1-byte variable (CTRL)
ALARM           BIT     CTRL.0              ; bit in a relocatable byte
SHUT            BIT     ALARM+1             ; the next bit
ENABLE_FLAG     BIT     60H                 ; an absolute bit
DONE_FLAG       BIT     24H.2               ; an absolute bit
P1_BIT2         EQU     90H.2               ; a SFR bit
RESTART         CODE    00H
INTVEC_0        CODE    RESTART + 3
INTVEC_1        CODE    RESTART + 0BH
INTVEC_2        CODE    RESTART + 1BH
SERBUF          DATA    SBUF                ; redfinition of the SFR SBUF
RESULT          DATA    40H
RESULT2         DATA    RESULT + 2
PORT1           DATA    90H                 ; a SFR symbol
BUFFER          IDATA   60H
BUF_LEN         EQU     20H
BUF_END         IDATA   BUFFER + BUF_LEN - 1
XSEG1           SEGMENT XDATA
RSEG            XSEG1

DTIM:           DS      6                   ;reserve 6-bytes for DTIM
TIME            XDATA   DTIM + 0
DATE            XDATA   DTIM + 3
```

LIT †

The **LIT** directive provides a simple text substitution facility. The **LIT** directive has the following format:

```
symbol        LIT       'literal string'
symbol        LIT       "literal string"
```

where

symbol is the name of the symbol to define. The literal string specified in the **LIT** directive will be substituted for each occurrence of ***symbol*** that is used in your assembly program.

literal string is a numeric expression which contains no forward references, or a simple relocatable expression.

Every time the ***symbol*** is encountered, it will be replaced by the ***literal string*** assigned to symbol name. The symbol name follows the same rules as other identifiers, that is, a literal name is not encountered if it not forms a separate token. If a substring is to be replaced, then ***symbol*** must be enclosed in braces: TEXT{***symbol***}. The assembler listing shows the expanded lines where literals are substituted.

Example

Source text containing literals before assembly:

```
                              $INCLUDE (REG51.INC)

                              REG1    LIT     'R1'
                              NUM     LIT     'A1'
                              DBYTE   LIT     "DATA BYTE"
                              FLAG    LIT     'ACC.3'

                              ?PR?MOD SEGMENT CODE
                                      RSEG    ?PR?MOD

                                      MOV     REG1,#5
                                      SETB    FLAG
                                      JB      FLAG,LAB_{NUM}
                                      PUSH    DBYTE 0
                              LAB_{NUM}:


                                      END
```

Assembler listing from previous example:

```
                          1     $INCLUDE (REG51.INC)
                    +1    80  +1 $RESTORE
                          81
                          82     REG1    LIT     'R1'
                          83     NUM     LIT     'A1'
                          84     DBYTE   LIT     "DATA BYTE"
                          85     FLAG    LIT     'ACC.3'
                          86
------                    87     ?PR?MOD SEGMENT CODE
------                    88             RSEG ?PR?MOD
                          89
000000 7E1005             90             MOV     R1,#5
000003 D2E3               91             SETB    ACC.3
000005 20E300     F       92             JB      ACC.3,LAB_A1
000008 C000               93             PUSH    DATA BYTE 0
00000A                    94     LAB_A1:
                          95
                          96             END
```

Memory Initialization

The memory initialization directives are used to initialize code or const space in either word, dword or byte units. The memory image starts at the point indicated by the current value of the location counter in the currently active segment.

DB

The **DB** directive initializes code memory with 8-bit byte values. The **DB** directive has the following format:

```
label:   DB  expression   ,  expression  …
```

where

label	is the symbol that is given the address of the initialized memory and
expression	is a byte value. Each ***expression*** may be a symbol, a character string, or an expression.

The **DB** directive can only be specified within a code or const segment. If the **DB** directive is used in a different segment, A251 will generate an error message.

Example

```
REQUEST:      DB    'PRESS ANY KEY TO CONTINUE', 0
TABLE:        DB    0,1,8,'A','0', LOW(TABLE),';'
ZERO:         DB    0, ''''
CASE_TAB:     DB    LOW(REQUEST), LOW(TABLE), LOW(ZERO)
```

DW

The **DW** directive initializes code memory with 16-bit word values. The **DW** directive has the following format:

```
label:   DW  expression   ,  expression  …
```

where

label is the symbol that is given the address of the initialized memory and

expression is the initialization data. Each ***expression*** may contain a symbol, a character string, or an expression.

The **DW** directive can only be specified within a code or const segment. If the **DW** directive is used in a different segment, A251 will generate an error message.

Example

```
TABLE:      DW   TABLE, TABLE + 10, ZERO
ZERO:       DW   0
CASE_TAB:   DW   CASE0, CASE1, CASE2, CASE3, CASE4
            DW   $
```


DD †

The **DD** directive initializes code memory with 32–bit double word values. The **DD** directive has the following format:

```
label:   DD  expression   ,  expression  …
```

where

label is the symbol that is given the address of the initialized memory and

expression is the initialization data. Each ***expression*** may contain a symbol, a character string, or an expression.

The **DD** directive can only be specified within a code or const segment. If the **DD** directive is used in a different segment, A251 will generate an error message.

Example

```
TABLE:      DD    TABLE, TABLE + 10, ZERO
            DD    $
ZERO:       DD    0
LONG_VAL:   DD    12345678H, 0FFFFFFFFH, 1
```

† New features in the A251 assembler and the MCS 251 architecture

Memory Reservation

The memory reservation directives are used to reserve space in either word, dword, byte, or bit units. The space reserved starts at the point indicated by the current value of the location counter in the currently active segment.

DBIT

The **DBIT** directive reserves space in a bit or ebit segment. The **DBIT** directive has the following format:

```
label:   DBIT  expression
```

where

label	is the symbol that is given the address of the reserved memory. The label is a symbol of the type BIT and gets the current address value and the memory class of the active segment. The label can only be used where a symbol of this type is allowed.
expression	is the number of bits to reserve. The ***expression*** cannot contain forward references, relocatable symbols, or external symbols.

The **DBIT** directive reserves space in the bit segment starting at the current address. The location counter for the bit segment is increased by the value of the ***expression***. You should note that the location counter for the bit segment references bits and not bytes.

NOTE
*The A251 assembler is a two–pass assembler. Symbols are collected and the length of each instruction is determined in the first pass. In the second pass, forward references are resolved and object code is produced. For these reasons, an expression used with the **DBIT** directive may not contain forward references.*

Example

```
ON_FLAG:      DBIT    1                                    ;reserve 1 bit
OFF_FLAG:     DBIT    1
```

DS

The **DS** directive reserves a specified number of bytes in a memory space. The **DS** directive has the following format:

```
label:   DS  expression
```

where

label	is the symbol that is given the address of the reserved memory. The label is a typeless number and gets the current address value and the memory class of the active segment. The label can only be used where a symbol of this type is allowed.
expression	is the number of bytes to reserve. The ***expression*** cannot contain forward references, relocatable symbols, or external symbols.

The **DS** directive reserves space in the current segment at the current address. The current address is then increased by the value of the ***expression***. The sum of the location counter and the value of the specified ***expression*** should not exceed the limitations of the current address space.

NOTE
*The A251 assembler is a two–pass assembler. Symbols are collected and the length of each instruction is determined in the first pass. In the second pass, forward references are resolved and object code is produced. For these reasons, an expression used with the **DS** directive may not contain forward references.*

Example

```
GAP:        DS    (($ + 16) AND 0FFF0H) – $
            DS    20
TIME:       DS    8
```

DSB †

The **DSB** directive reserves a specified number of bytes in a memory space. The **DSB** directive has the following format:

```
label:   DSB  expression
```

† New features in the A251 assembler and the MCS 251 architecture

where

label is the symbol that is given the address of the reserved memory. The label is a symbol of the type BYTE and gets the current address value and the memory class of the active segment. The label can only be used where a symbol of this type is allowed.

expression is the number of bytes to reserve. The ***expression*** cannot contain forward references, relocatable symbols, or external symbols.

The **DSB** directive reserves space in the current segment at the current address. The current address is then increased by the value of the *expression*. The sum of the location counter and the value of the specified *expression* should not exceed the limitations of the current address space.

NOTE
*The A251 assembler is a two–pass assembler. Symbols are collected and the length of each instruction is determined in the first pass. In the second pass, forward references are resolved and object code is produced. For these reasons, an expression used with the **DSB** directive may not contain forward references.*

Example

```
DAY:      DSB    1
MONTH:    DSB    1
HOUR:     DSB    1
MIN:      DSB    1
```

DSW †

The **DSW** directive reserves a specified number of words in a memory space. The **DSW** directive has the following format:

```
label:   DSW  expression
```

where

label is the symbol that is given the address of the reserved memory. The label is a symbol of the type WORD and gets the current address value and the memory class of the active

segment. The label can only be used where a symbol of this type is allowed.

expression is the number of bytes to reserve. The ***expression*** cannot contain forward references, relocatable symbols, or external symbols.

The **DSW** directive reserves space in the current segment at the current address. The current address is then increased by the value of the *expression*. The sum of the location counter and the value of the specified *expression* should not exceed the limitations of the current address space.

NOTE
*The A251 assembler is a two–pass assembler. Symbols are collected and the length of each instruction is determined in the first pass. In the second pass, forward references are resolved and object code is produced. For these reasons, an expression used with the **DSW** directive may not contain forward references.*

Example

```
YEAR:          DSW   1
DAYinYEAR:     DSW   1
```

DSD †

The **DSD** directive reserves a specified number of double words in a memory space. The **DSD** directive has the following format:

```
label:   DSD  expression
```

where

label is the symbol that is given the address of the reserved memory. The label is a symbol of the type DWORD and gets the current address value and the memory class of the active segment. The label can only be used where a symbol of this type is allowed.

expression is the number of bytes to reserve. The ***expression*** cannot contain forward references, relocatable symbols, or external symbols.

† New features in the A251 assembler and the MCS 251 architecture

The **DSD** directive reserves space in the current segment at the current address. The current address is then increased by the value of the ***expression***. The sum of the location counter and the value of the specified ***expression*** should not exceed the limitations of the current address space.

NOTE
*The A251 assembler is a two–pass assembler. Symbols are collected and the length of each instruction is determined in the first pass. In the second pass, forward references are resolved and object code is produced. For these reasons, an expression used with the **DSD** directive may not contain forward references.*

Example

```
SEC_CNT:        DSD     1
LONG_ARR:       DSD     50
```

4

Procedure Declaration †

A251 provides procedures to implement the concept of subroutines. Procedures can be executed in-line (control "falls through" to them), jumped to, or invoked by a CALL. Calls are recommended as a better programming practice.

PROC / ENDP †

The PROC / ENDP directive pair is used to define a label for a sequence of machine instructions called a procedure. A procedure is called within the same physical 64KByte segment (LCALL or ACALL = NEAR) or from a different 64kbyte segment (ECALL = FAR). A procedure may have either the type NEAR or FAR. Unless procedures known from high level languages, the scoping of identifiers is different in the assembly language. Identifiers must be unique in A251 because the visibility is module wide. The format of the PROC/ENDP directives is:

```
name        PROC    [ type ]
.
.
.
            ; procedure text
.
.
.
```

```
            RET
name        ENDP
```

where

name is the name of the procedure.

type specifies the type of the procedure, and must be one of the following:

Type	Description
none	The type defaults to NEAR
NEAR	Defines a **near** procedure; called with LCALL or ACALL.
FAR	Defines a **far** procedure; called with ECALL.

You should specify FAR if the procedure will be called from different 64KByte segment. A procedure normally ends with a RET instruction. The software instruction RET will be automatically converted to an appropriate machine return instruction, that is:

RET from a near procedure.

ERET from a far procedure.

Example

```
P100                PROC    NEAR
                    RET                 ; near return
                    ENDP

P200                PROC    FAR
                    RET                 ; far return (ERET)
                    ENDP

P300                PROC    NEAR
                    CALL    P100        ; LCALL
                    CALL    P200        ; ECALL
                    RET                 ; near return
                    ENDP

                    END
```

LABEL †

A label is a symbol name for an address location in a segment. The LABEL directive can be used to define a program label. The label name can be followed by a colon, but it is not required. The label inherits the attributes of the program or code segment currently active. The LABEL directive may therefore never be used outside the scope of a program segment. The syntax of the LABEL directive is:

```
name[:]       LABEL      [ type ]
```

where

name is the name of the label.

type specifies the type of the label, and must be one of the following:

Type	Description
none	The type defaults to NEAR
NEAR	Defines a **near** label.
FAR	Defines a **far** label; use ECALL or EJMP.

You should specify FAR if the label will be referenced from a different 64KByte segment. NEAR lets you refer to this label for the current 64KByte segment.

Example

```
                RSEG    ECODE_SEG1      ; activate an ECODE segment
ENTRY:          LABEL   FAR             ; entry point

                RSEG    ECODE_SEG2      ; activate another ECODE segment
                EJMP    ENTRTY          ; Jump across 64KB segment
```

Program Linkage

Program linkage directives allow the separately assembled modules to communicate by permitting inter-module references and the naming of modules.

PUBLIC

The **PUBLIC** directive lists symbols that may be used in other object modules. The **PUBLIC** directive makes the specified symbols available in the generated object module. This, in effect, publicizes the names of these symbols. The **PUBLIC** directive has the following format:

```
PUBLIC  symbol   ,  symbol  …
```

where

symbol must be a symbol that was defined somewhere within the source file. Forward references to symbol names are permitted. All symbol names, with the exception of register symbols and segment symbols, may be specified with the **PUBLIC** directive. Multiple symbols must be separated with a comma (**,**).

If you want to use public symbols in other source files, the **EXTRN** or **EXTERN** directive must be used to specify that the symbols are declared in another object module.

Example

```
PUBLIC    PUT_CRLF, PUT_STRING, PUT_EOS
PUBLIC    ASCBIN, BINASC
PUBLIC    GETTOKEN, GETNUMBER
```

EXTRN / EXTERN

The **EXTRN** or **EXTERN** † directive lists symbols that are referenced in the current source module that are actually declared in other modules. The format for the **EXTRN** / **EXTERN** directive is as follows:

† New features in the A251 assembler and the MCS 251 architecture

```
EXTRN     class   : type   (symbol   ,  symbol  … )
EXTERN    class   : type   (symbol   ,  symbol  … )     †
```

where

class	is the memory class where the symbol has been defined and may be one of the following: **BIT**, **CODE**, **CONST** †, **DATA**, **EBIT** †, **ECONST** †, **EDATA** †, **ECODE** †, **HDATA** †, **HCONST** †, **IDATA**, **XDATA**, or **NUMBER** (to specify a typeless symbol).
type †	is the symbol type of the external symbol and may be one of the following: **BYTE**, **WORD**, **DWORD**, **NEAR**, **FAR.**
symbol	is an external symbol name.

The **EXTRN** or **EXTERN** directive may appear anywhere in the source program. Multiple symbols may be separated and included in parentheses following the class and type information.

Symbol names that are specified with the **EXTRN** / **EXTERN** directive must have been specified as public symbols with the **PUBLIC** directive in the source file in which they were declared.

The Linker/Locator resolves all external symbols at link time and verifies that the symbol class and symbol types (specified with the **EXTRN** / **EXTERN** and **PUBLIC** directives) matches. Symbols with the class **NUMBER** matches to every memory class.

Examples

```
EXTRN     CODE (PUT_CRLF), DATA (BUFFER)
EXTERN    CODE (BINASC, ASCBIN)
EXTRN     NUMBER (TABLE_SIZE)
EXTERN    CODE:FAR (main)  †
EXTRN     EDATA:BYTE (VALUE, COUNT)  †
EXTRN     NCONST:DWORD (LIMIT)  †
```

NAME

The **NAME** directive specifies the name to use for the object module generated for the current program. The filename for the object file is not the object module name. The object module name is embedded within the object file. The format for the **NAME** directive is as follows:

```
NAME  modulename
```

where

modulename is the name to use for the object module and can be up to 40 characters long. The modulename must adhere to the rules for symbol names.

If a **NAME** directive is not present in the source program, the object module name will be the *basename* of the source file without the extension.

NOTE
*Only one **NAME** directive may be specified in a source file.*

Example

```
NAME      PARSERMODULE
```


Address Control

The following directives allow the control of the address location counter or the control of absolute register symbols.

ORG

The **ORG** directive is used to alter the location counter of the current active segment and sets an new origin for statements that follow the directive. The format for the **ORG** statement is as follows:

```
ORG  expression
```

where

expression must be an absolute or simple relocatable expression and may not have any forward references. Only absolute addresses or symbol values of the current segment may be used.

When an **ORG** statement is encountered, the assembler calculates the value of the *expression* and changes the location counter for the current segment. If the **ORG** statement occurs in an absolute segment, the location counter will be

assigned the absolute address value. If the **ORG** statement occurs in a relocatable segment, the location counter will be assigned the offset of the specified *expression*.

The **ORG** directive changes the location counter but does not produce a new segment. A possible address gap may be introduced into the current segment. With absolute segments, the location counter cannot reference an address prior to the segment base.

NOTE
*The A251 assembler is a two–pass assembler. Symbols are collected and the length of each instruction is determined in the first pass. In the second pass, forward references are resolved and object code is produced. For these reasons, an expression used with the **ORG** directive may not contain forward references.*

Example

```
ORG    100H
ORG    RESTART
ORG    EXTI1
ORG    ($ + 16) AND 0FFF0H
```

EVEN †

The **EVEN** directive ensures that code or data following EVEN is aligned on a word boundary. The assembler creates a gap of one byte if necessary. The content of the byte gap is undefined. The **EVEN** directive has the following syntax:

```
EVEN
```

Example

```
MYDATA  SEGMENT DATA WORD   ; word alignment
        RSEG    MYDATA      ; activate segment
var1:   DSB     1           ; reserve a byte variable
        EVEN                ; ensure word alignment
var2:   DSW     1           ; reserve a word variable
```

USING

The **USING** directive specifies which register bank to use for coding the **AR0** through **AR7** registers. The **USING** directive is specified as follows:

```
USING  expression
```

where

expression is the register bank number which must be a value between 0 and 3.

The **USING** directive does not generate any code to change the register bank. Your program must make sure the correct register bank is selected. For example, the following code can be used to select register bank 2:

```
                PUSH   PSW                ;save PSW/register bank
                MOV    PSW,#(2 SHL 3)     ;select register bank 2
.
.
.
                ;function or subroutine body
.
.
.
                POP    PSW                ;restore PSW/register bank
```

The register bank selected by the **USING** directive is marked in the object file and the memory area required by these registerbank reserved by the Linker/Locator.

The value of **AR0** through **AR7** is calculated as the absolute address of **R0** through **R7** in the register bank specified by the **USING** directive. Some 8051 instruction (i.e. PUSH / POP) only allow to use absolute register addresses. By default the register bank 0 is assigned to the symbols **AR0** through **AR7**.

NOTE
*When the **EQU** directive is used to define a symbol for an **AR**n register, the address of the register **R**n is calculated when the symbol is defined; not when it is used. If the **USING** directive subsequently changes the register bank, the defined symbol will not have the proper address of the **AR**n register and the generated code is likely to fail.*

Example

```
        USING     3
        PUSH      AR2                       ; Push register 2 in bank 3

        USING     1
        PUSH      AR2                       ; Push register 2 in bank 1
```

Other Directives

END

The **END** directive signals the end of the assembly module. Any text in the assembly file that appears after the **END** directive is ignored.

The **END** directive is required in every assembly source file. If the **END** statement is excluded, A251 will generate a fatal error message.

Example

```
END
```

4

Chapter 5. Standard Macros

A macro is a name that you assign to one or more assembly statements. A251 provides a macro processor that enables you to define and to use macros in your 8051 assembly programs. This chapter describes some of the features and advantages of using macros, lists the directives and operators that are used in macro definitions, and provides a number of example macros.

When you define a macro, you provide text (usually assembly code) that you want to associate with a macro name. Then, when you want to include the macro text in your assembly program, you provide the name of the macro. The A251 assembler will replace the macro name with the text specified in the macro definition.

Macros provide a number of advantages when writing assembly programs.

- The frequent use of macros can reduce programmer induced errors. A macro allows you to define instruction sequences that are used repetitively throughout your program. Subsequent use of the macro will faithfully provide the same results each time. A macro can help reduce the likelihood of errors introduced in repetitive programming sequences. Of course, introduction of an error into a macro definition will cause that error to be duplicated where the macro is used.
- The scope of symbols used in a macro is limited to that macro. You do not need to be concerned about utilizing a previously used symbol name.
- Macros are well suited for the creation of simple code tables. Production of these tables by hand is both tedious and error prone.

A macro can be thought of as a subroutine call with the exception that the code that would be contained in the subroutine is included in–line at the point of the macro call. However, macros should not be used to replace subroutines. Each invocation of a subroutine only adds code to call the subroutine. Each invocation of a macro causes the assembly code associated with the macro to be included in–line in the assembly program. This can cause a program to grow rapidly if a large macro is used frequently. In a static environment, a subroutine is the better choice, since program size can be considerably reduced. But in time critical, dynamic programs, macros will speed the execution of algorithms or other frequently called statements without the penalty of the procedure calling overhead.

Use the following guidelines when deciding between macros or subroutines:

- Subroutines are best used when certain procedures are frequently executed or when memory space usage must be kept to a minimum.
- Macros should be used when maximum processor speed is required and when memory space used is of less importance.
- Macros can also be used to make repetitive, short assembly blocks more convenient to enter.

Directives

A251 provides a number of directives that are used specifically for defining macros. These directives are listed in the following table:

Directive	Description
ENDM	Ends a macro definition.
EXITM	Causes the macro expansion to immediately terminate.
IRP	Specifies a list of arguments to be substituted, one at a time, for a specified parameter in subsequent lines.
IRPC	Specifies an argument to be substituted, one character at a time, for a specified parameter in subsequent lines.
LOCAL	Specifies up to 16 local symbols used within the macro.
MACRO	Begins a macro definition and specifies the name of the macro and any parameters that may be passed to the macro.
REPT	Specifies a repetition factor for subsequent lines in the macro.

Refer to “Assembler Controls” on page 115 as well as the following sections for more information on these and other directives.

Defining a Macro

Macros must be defined in the program before they can be used. A macro definition begins with the **MACRO** directive which declares the name of the macro as well as the formal parameters. The macro definition must be terminated with the **ENDM** directive. The text between the **MACRO** and **ENDM** directives is called the macro body.

Example

```
WAIT                MACRO      X          ; macro definition
                      REPT     X          ; generate X NOP instructions
                      NOP
                      ENDM                ; end REPT
                    ENDM                  ; end MACRO
```

In this example, **WAIT** is the name of the macro and **X** is the only formal parameter.

In addition to the **ENDM** directive, the **EXITM** directive can be used to immediately terminate a macro expansion. When an **EXITM** directive is detected, the macro processor stops expanding the current macro and resumes processing after the next **ENDM** directive. The **EXITM** directive is useful in conditional statements.

Example

```
WAIT                MACRO          X      ; macro definition
                      IF NUL X             ; make sure X has a value
                      EXITM                ; if not then exit
                      ENDIF

                      REPT         X      ; generate X NOP instructions
                      NOP
                      ENDM                 ; end REPT
                    ENDM                   ; end MACRO
```

5

Parameters

Up to 16 parameters can be passed to a macro in the invocation line. Formal parameter names must be defined using the **MACRO** directive.

Example

```
MNAME  MACRO  P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,P12,P13,P14,P15,P16
```

defines a macro with 16 parameters. Parameters must be separated by commas both in the macro definition and invocation. The invocation line for the above macro would appear as follows:

```
MNAME A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P
```

where **A**, **B**, **C**, ... **O**, **P** are parameters that correspond to the format parameter names **P1**, **P2**, **P3**, ... **P15**, **P16**.

Null parameters can be passed to a macro. Null parameters have the value NULL and can be tested for using the **NUL** operator described later in this chapter. If a parameter is omitted from the parameter list in the macro invocation, that parameter is assigned a value of NULL.

Example

```
MNAME A,,C,,E,,G,,I,,K,,M,,O,
```

P2, **P4**, **P6**, **P8**, **P10**, **P12**, **P14**, and **P16** will all be assigned the value NULL when the macro is invoked. You should note that there are no spaces between the comma separators in the above invocation line. A space has an ASCII value of 20h and is not equivalent to a NULL.

Labels

You can use labels within a macro definition. By default, labels used in a macro are global and if the macro is used more than once in a module, A251 will generate an error.

Example

```
MS-DOS MACRO ASSEMBLER A251
OBJECT MODULE PLACED IN M_GLAB.OBJ
ASSEMBLER INVOKED BY:  A251 M_GLAB.A251


LOC  OBJ            LINE     SOURCE

                       1     GLABEL  MACRO
                       2     LOOP:   NOP
                       3             JMP     LOOP
                       4             ENDM
                       5
                       6
                       7             GLABEL
0000 00                8+1   LOOP:   NOP
0001 80FD              9+1           JMP     LOOP
                      10             GLABEL
                      11+1   LOOP:   NOP
***  ____________________^
*** ERROR #9, LINE #11, ATTEMPT TO DEFINE AN ALREADY DEFINED LABEL
0003 80FB             12+1           JMP     LOOP
                      13
                      14
                      15     END
```

Labels used in a macro should be local labels. Local labels are visible only within the macro and will not generate errors if the macro is used multiple times

in one source file. You can define a label (or any symbol) used in a macro to be local with the **LOCAL** directive. Up to 16 local symbols may be defined using the **LOCAL** directive.

Example

```
CLRMEM              MACRO     ADDR, LEN
                    LOCAL     LOOP
                    MOV       R7, #LEN
                    MOV       R0, #ADDR
                    MOV       A, #0
LOOP:               MOV       @R0, A
                    INC       R0
                    DJNZ      R7, LOOP
                    ENDM
```

In this example, the label **LOOP** is local because it is defined with the **LOCAL** directive. Any symbol that is not defined using the **LOCAL** directive will be a global symbol.

A251 generates an internal symbol for local symbols defined in a macro. The internal symbol has the form ??0000 and is incremented each time the macro is invoked. Therefore, local labels used in a macro are unique and will not generate errors.

5

Repeating Blocks

A251 provides the ability to repeat a block of text within a macro. The **REPT**, **IRP**, and **IRPC** directives are used to specify text to repeat within a macro. Each of these directives must be terminated with an **ENDM** directive.

REPT

The **REPT** directive will cause a block of text to be repeated a fixed number of times. The following macro:

```
DELAY               MACRO                ;macro definition
                      REPT     5         ;insert 5 NOP instructions
                      NOP
                      ENDM                ;end REPT block
                    ENDM                  ;end macro definition
```

will insert 5 NOP instructions when it is invoked.

Example

```
NOP
NOP
NOP
NOP
NOP
```

IRP

The **IRP** directive will repeat a block once for each argument in a specified list. A specified parameter in the text block will be replaced by each argument. The following macro:

```
CLRREGS     MACRO                                          ; macro definition
               IRP   RNUM, <R0,R1,R2,R3,R4,R5,R6,R7>
                     MOV   RNUM, #0
               ENDM                                        ; end IRP
            ENDM                                           ; end MACRO
```

replaces the argument **RNUM** with **R0**, **R1**, **R2**, … **R7** and will generate the following code when invoked:

```
MOV    R0, #0
MOV    R1, #0
MOV    R2, #0
MOV    R3, #0
MOV    R4, #0
MOV    R5, #0
MOV    R6, #0
MOV    R7, #0
```

IRPC

The **IRPC** directive will repeat a block once for each character in the specified argument. A specified parameter in the text block will be replaced by each character. The following macro:

```
DEBUGOUT          MACRO                    ; macro definition
                  IRPC      CHR, <TEST>
                  JNB       TI, $          ; wait for xmitter
                  CLR       TI
                  MOV       A,#'CHR'
                  MOV       SBUF,A         ; xmit CHR
                  ENDM                     ; end IRPC
                  ENDM                     ; end MACRO
```

replaces the argument **CHR** with the characters **T**, **E**, **S**, and **T** and will generate the following code when invoked:

```
JNB   TI, $   ; WAIT FOR XMITTER
CLR   TI
MOV   A,#'T'
MOV   SBUF,A  ; XMIT T
JNB   TI, $   ; WAIT FOR XMITTER
CLR   TI
MOV   A,#'E'
MOV   SBUF,A  ; XMIT E
JNB   TI, $   ; WAIT FOR XMITTER
CLR   TI
MOV   A,#'S'
MOV   SBUF,A  ; XMIT S
JNB   TI, $   ; WAIT FOR XMITTER
CLR   TI
MOV   A,#'T'
MOV   SBUF,A  ; XMIT T
```

Nested Definitions

Macro definitions can be nested up to nine levels deep.

Example

```
L1                MACRO
                  LOCAL     L2
                  L2        MACRO
                            INC   R0
                            ENDM
                  MOV       R0, #0
                  L2
                  ENDM
```

The macro **L2** is defined within the macro definition of **L1**. Since the **LOCAL** directive is used to define **L2** as a local symbol, it is not visible outside **L1**. If you want to use **L2** outside of **L1**, exclude **L2** from the **LOCAL** directive symbol list.

Invocation of the **L1** macro generates the following:

```
MOV    R0, #0
INC    R0
```

Nested Repeating Blocks

You can also nest repeating blocks, specified with the **REPT**, **IRP**, and **IRPC** directives.

Example

```
PORTOUT             MACRO                       ; macro definition
                      IRPC      CHR, <Hello>
                        REPT    4               ; wait for 4 cycles
                        NOP
                        ENDM                    ; end REPT
                      MOV       A,#'CHR'
                      MOV       P0,A            ; write CHR to P0
                      ENDM                      ; end IRPC
                    ENDM                        ; end MACRO
```

This macro nests a **REPT** block within an **IRPC** block.

Recursive Macros

Macros can call themselves directly or indirectly (via another macro). However, the total number of levels of recursion may not exceed nine. A fatal error will be generated if the total nesting level is greater than nine. The following example shows a recursive macro that is invoked by a non–recursive macro.

```
RECURSE             MACRO       X           ; recursive macro
IF X<>0
                    RECURSE     %X-1
                    ADD         A,#X        ; gen add a,#?
ENDIF
                    ENDM

SUMM                MACRO       X           ; macro to sum numbers
                    MOV         A,#0        ; start with zero
IF NUL X                                    ; exit if null argument
                    EXITM
ENDIF
IF X=0                                      ; exit if 0 argument
                    EXITM
ENDIF

                    RECURSE X               ; sum to 0
                    ENDM
```

Operators

A251 provides a number of operators that may be used within a macro definition. The following table lists the operators and gives a description of each.

Operator	Description
NUL	The NUL operator can be used to determine if a macro argument is NULL. NUL generates a non–zero value if its argument is a NULL. Non–NULL arguments will generate a value of 0. The NUL operator can be used with an IF control to enable condition macro assembly.
&	The ampersand character is used to concatenate text and parameters.
< >	Angle brackets are used to literalize delimiters like commas and blanks. Angle brackets are required when passing these characters to a nested macro. One pair of angle brackets is required for every nesting level.
%	The percent sign is used to prefix a macro argument that should be interpreted as an expression. When this operator is used, the numeric value of the following expression is calculated. That value is passed to the macro instead of the expression text.
;;	A double semicolon indicates that subsequent text on the line should be ignored. The remaining text is not processed or emitted. This helps to reduce memory usage.
!	If an exclamation mark is used in front of a character, that character will be literalized. This allows character operators to be passed to a macro as a parameter.

NUL Operator

When a formal parameter in a macro call is omitted, the parameter is given a value of NULL. You can check for NULL parameters by using the **NUL** operator within an **IF** control statement in the macro. The **NUL** operator requires an argument. If no argument is found, **NUL** returns a value of 0 to the IF control.

For example, the following macro definition:

```
EXAMPLE      MACRO   X
  IF NUL X
    EXITM
  ENDIF
ENDM
```

when invoked by:

```
EXAMPLE
```

will cause the **IF NUL X** test to pass, process the **EXITM** statement, and exit the macro expansion.

NOTE
A blank character (‘ ’) has an ASCII value of 20h and is not equivalent to a NULL.

& Operator

The ampersand macro operator (**&**) can be used to concatenate text and macro parameters. The following macro declaration demonstrates the proper use of this operator.

```
MAK_NOP_LABEL              MACRO    X
LABEL&X:                   NOP
                           ENDM
```

The **MAK_NOP_LABEL** macro will insert a new label and a NOP instruction for each invocation. The argument will be appended to the text **LABEL** to form the label for the line.

Example

```
LOC  OBJ            LINE     SOURCE

                       1     MAK_NOP_LABEL   MACRO   X
                       2     LABEL&X:        NOP
                       3     ENDM
                       4
                       5
                       6     MAK_NOP_LABEL 1
0000 00                7+1   LABEL1:         NOP
                       8     MAK_NOP_LABEL 2
0001 00                9+1   LABEL2:         NOP
                      10     MAK_NOP_LABEL 3
0002 00               11+1   LABEL3:         NOP
                      12     MAK_NOP_LABEL 4
0003 00               13+1   LABEL4:         NOP
                      14
                      15     END
```

The **MAK_NOP_LABEL** macro is invoked in the above example in lines 6, 8, 10, and 12. The generated label and NOP instructions are shown in lines 7, 9, 11, and 13. Note that the labels are concatenated with the argument that is passed in the macro invocation.

< and > Operators

The angle bracket characters (< >) are used to enclose text that should be passed literally to macros. Some characters; for example, the comma; cannot be passed without being enclosed within angle brackets.

The following example shows a macro declaration and invocation passing an argument list within angle brackets.

```
                    1     FLAG_CLR        MACRO   FLAGS
                    2                     MOV     A, #0
                    3                       IRP   F, <FLAGS>
                    4                       MOV   FLAG&F, A
                    5                       ENDM
                    6                     ENDM
                    7
                    8     DSEG
0000                9     FLAG1:  DS 1
0001               10     FLAG2:  DS 1
0002               11     FLAG3:  DS 1
0003               12     FLAG4:  DS 1
0004               13     FLAG5:  DS 1
0005               14     FLAG6:  DS 1
0006               15     FLAG7:  DS 1
0007               16     FLAG8:  DS 1
0008               17     FLAG9:  DS 1
                   18
                   19     CSEG
                   20
                   21             FLAG_CLR        <1>
0000 7400          22+1                   MOV     A, #0
                   23+1                     IRP   F, <1>
                   24+1                     MOV   FLAG&F, A
                   25+1                     ENDM
0002 F500          26+2                     MOV   FLAG1, A
                   27             FLAG_CLR        <1,2,3>
0004 7400          28+1                   MOV     A, #0
                   29+1                     IRP   F, <1,2,3>
                   30+1                     MOV   FLAG&F, A
                   31+1                     ENDM
0006 F500          32+2                     MOV   FLAG1, A
0008 F501          33+2                     MOV   FLAG2, A
000A F502          34+2                     MOV   FLAG3, A
                   35             FLAG_CLR        <1,3,5,7>
000C 7400          36+1                   MOV     A, #0
                   37+1                     IRP   F, <1,3,5,7>
                   38+1                     MOV   FLAG&F, A
                   39+1                     ENDM
000E F500          40+2                     MOV   FLAG1, A
0010 F502          41+2                     MOV   FLAG3, A
0012 F504          42+2                     MOV   FLAG5, A
0014 F506          43+2                     MOV   FLAG7, A
.
.
```

5

In the above example, the **FLAG_CLR** macro is declared to clear any of a number of flag variables. The **FLAGS** argument specifies a list of arguments that are used by the **IRP** directive in line 3. The **IRP** directive repeats the instruction **MOV FLAG&F, A** for each parameter in the **FLAGS** argument.

The **FLAG_CLR** macro is invoked in lines 21, 27, and 35. In line 21, only one parameter is passed. In line 27, three parameters are passed, and in line 35, four parameters are passed. The parameter list is enclosed in angle brackets so that it may be referred to as a single macro parameter, **FLAGS**. The code generated by the macro is found in lines 26, 32–34, and 40–43.

% Operator

The percent character (**%**) is used to pass the value of an expression to a macro rather than passing the literal expression itself. For example, the following program example shows a macro declaration that requires a numeric value along with macro invocations that use the percent operator to pass the value of an expression to the macro.


```
            1 OUTPORT MACRO   N
            2         MOV     A, #N
            3         MOV     P0, A
            4         ENDM
            5
            6
  00FF      7 RESET_SIG       EQU     0FFh
  0000      8 CLEAR_SIG       EQU     0
            9
           10
           11         OUTPORT %(RESET_SIG AND NOT 11110000b)
0000 740F  12+1       MOV     A, #15
0002 F580  13+1       MOV     P0, A
           14
           15         OUTPORT %(CLEAR_SIG OR 11110000b)
0004 74F0  16+1       MOV     A, #240
0006 F580  17+1       MOV     P0, A
```

In this example, the expressions evaluated in lines 11 and 15 could not be passed to the macro because the macro expects a numeric value. Therefore, the expressions must be evaluated before the macro. The percent sign forces A251 to generate a numeric value for the expressions. This value is then passed to the macro.

;; Operator

The double semicolon operator is used to signal that the remaining text on the line should not be emitted when the macro is expanded. This operator is typically used to precede comments that do not need to be expanded when the macro is invoked.

Example

```
REGCLR              MACRO     CNT
REGNUM              SET       0
                    MOV       A, #0            ;; load A with 0
                      REPT    CNT              ;; rpt for CNT registers
                      MOV     R&REGNUM, A      ;; set R# to 0
                      REGNUM SET %(REGNUM+1)
                      ENDM
                    ENDM
```

! Operator

The exclamation mark operator is used to indicate that a special character is to be passed literally to a macro. This operator enables you to pass comma and angle bracket characters, that would normally be interpreted as delimiters, to a macro.

5

Invoking a Macro

Once a macro has been defined, it can be called many times in the program. A macro call consists of the macro name plus any parameters that are to be passed to the macro.

In the invocation of a macro, the position of the actual parameters corresponds to the position of the parameter names specified in the macro definition. A251 performs parameter substitution in the macro starting with the first parameter. The first parameter passed in the invocation replaces each occurrence of the first formal parameter in the macro definition, the second parameter that is passed replaces the second formal parameter in the macro definition, and so on.

If more parameters are specified in the macro invocation than are actually declared in the macro definition, A251 ignores the additional parameters. If fewer parameters are specified than declared, A251 replaces the missing parameters with a NULL character.

To invoke a macro in your assembly programs, you must first define the macro. For example, the following definition:

```
.
.
.
DELAY                MACRO     CNT    ;macro definition
                       REPT    CNT    ;insert CNT NOP instructions
                       NOP
                       ENDM           ;end REPT block
                     ENDM             ;end macro definition
.
.
.
```

defines a macro called **DELAY** that accepts one argument **CNT**. This macro will generate **CNT** NOP instructions. So, if **CNT** is equal to 3, the emitted code will be:

```
NOP
NOP
NOP
```

The following code shows how to invoke the **DELAY** macro from an assembly program.

```
.
.
.
LOOP:                MOV         P0, #0        ;clr PORT 0
                     DELAY       5             ;wait 5 NOPs
                     MOV         P0, #0ffh     ;set PORT 0
                     DELAY       5             ;wait 5 NOPs
                     JMP         LOOP          ;repeat
.
.
.
```

In this example, a value of 0 is written to port 0. The **DELAY** macro is then invoked with the parameter 5. This will cause 5 NOP instructions to be inserted into the program. A value of 0FFh is written to port 0 and the **DELAY** macro is invoked again. The program then repeats.

Chapter 6. Macro Processing Language

The Macro Processing Language (MPL) is a string replacement facility. The macro processing language is enabled with the assembler control MPL and fully compatible to the Intel ASM 51 macro processing language. It permits you to write repeatedly used sections of code once and then insert that code at several places in your program. Perhaps MPL's most valuable capability is conditional assembly-with all microprocessors, compact configuration dependent code is very important to good program design. Conditional assembly of sections of code can help to achieve the most compact code possible.

Overview

The MPL processor views the source file in different terms than the assembler: to the assembler, the source file is a series of lines – control lines, and directive lines. To the MPL processor, the source file is a long string of characters.

All MPL processing of the source file is performed before your code is assembled. Because of this independent processing of the MPL macros and assembly of code, we must differentiate between macro-time and assembly-time. At macro-time, assembly language symbols and labels are unknown. SET and EQU symbols, and the location counter are also not known. Similarly, at assembly-time, no information about the MPL is known.

The MPL processor scans the source file looking for macro calls. A macro call is a request to the processor to replace the macro name of a built-in or user-defined macro by some replacement text.

Creating and Calling MPL Macros

The MPL processor is a character string replacement facility. It searches the source file for a macro call, and then replaces the call with the macro's return value. A % character signals a macro call.

The MPL processor function DEFINE creates macros. MPL processor functions are a predefined part of the macro language, and can be called without definition. The syntax for DEFINE is:

```
%[*]DEFINE (macro name) [parameter-list] (macro-body)
```

DEFINE is the most important macro processor function. Each of the symbols in the syntax above (macro name, parameter-list, and macro-body) are described in the following.

Creating Parameterless Macros

When you create a parameterless macro, there are two parts to a DEFINE call:

- **macro name**
  The macro name defines the name used when the macro is called.
- **macro body**
  The macro-body defines the return value of the call.

The syntax of a **parameterless macro** definition is shown below:

```
%*DEFINE (macro name) (macro-body)
```

The ‘ %’ is the metacharacter that signals a macro call. The ‘*’ is the literal character. The use of the literal character is described later in this part.

Macro names have the following conventions:

- Maximum of 31 characters long
- First character: ‘A’ - ‘Z’, ‘a’ - ‘z’, ‘_’, or ‘?’
- Other characters: ‘A’ - ‘Z’, ‘a’ - ‘z’, ‘_’, ‘?’, ‘0’ - ‘9’

The macro-body is usually the replacement text of the macro call. However, the macro-body may contain calls to other macros. If so, the replacement text is actually the fully expanded macro-body, including the calls to other macros. When you define a macro using the syntax shown above, macro calls contained in the body of the macro are not expanded, until you call the macro.

The syntax of DEFINE requires that left and right parentheses surround the macro-body. For this reason, you must have balanced parentheses within the macro-body (each left parenthesis must have a succeeding right parenthesis, and each right parenthesis must have a preceding left parenthesis). We call character strings that meet these requirements balanced-text.

To call a macro, use the metacharacter followed by the macro name for the MPL macro. (The literal character is not needed when you call a user-defined macro.) The MPL processor will remove the call and insert the replacement text of the call. If the macro- body contains any call to other macros, they will be replaced with their replacement text.

Once a macro has been created, it may be redefined by a second DEFINE.

MPL Macros with Parameters

Parameters in a macro body allow to fill in values when you call the MPL macro. This permits you to design a generic macro that produces code for many operations.

The term parameter refers to both the formal parameters that are specified when the macro is defined, and the actual parameters or arguments that are replaced when the macro is called.

The syntax for defining MPL macros with parameters is:

```
%*DEFINE (macro-name(parameter-list)) (macro-body)
```


The parameter-list is a list of identifiers separated by macro delimiters. The identifier for each parameter must be unique.

Typically, the macro delimiters are parentheses and commas. When using these delimiters, you would enclose the parameter-list in parentheses and separate each formal parameter with a comma. When you define a macro using parentheses and commas as delimiters, you must use those same delimiters, when you call that macro.

The macro-body must be a balanced-text string. To indicate the locations of parameter replacement, place the parameter's name preceded by the metacharacter in the macro-body. The parameters may be used any number of times and in any order within the macro-body. If a macro has the same name as

one of the parameters, the macro cannot be called within the macro-body since this would lead to infinite recursion.

The example below shows the definition of a macro with three dummy parameters - SOURCE, DESTINATION, and COUNT. The macro will produce code to copy any number of bytes from one part of memory to another.

```
%*DEFINE (BMOVE (src, dst, cnt)) LOCAL lab  (
      MOV  R0,#%src
      MOV  R1,#%dst
      MOV  R2,#%cnt
%lab: MOV  A,@R0
      MOV  @R1,A
      INC  R0
      INC  R1
      DJNZ R2, %lab
)
```

To call the above macro, you must use the metacharacter followed by the macro's name similar to simple macros without parameters. However, a list of the actual parameters must follow. The actual parameters must be surrounded in the macro definition. The actual parameters must be balanced-text and may optionally contain calls to other macros. A simple program example with the macro defined above might be:

Assembler source text

```
%*DEFINE (BMOVE (src, dst, cnt)) LOCAL lab  (
      MOV  R0,#%src
      MOV  R1,#%dst
      MOV  R2,#%cnt
%lab: MOV  A,@R0
      MOV  @R1,A
      INC  R0
      INC  R1
      DJNZ R2, %lab
)

ALEN  EQU  10       ; define the array size
DSEC  SEGMENT IDATA ; define a IDATA segment
PSEC  SEGMENT CODE  ; define a CODE segment

      RSEG DSEC     ; activate IDATA segment
arr1: DS   ALEN     ; define arrays
arr2: DS   ALEN

      RSEG PSEC     ; activate CODE segment
; move memory block
%BMOVE (arr1,arr2,ALEN)

      END
```

The following listing shows the assembler listing of the above source code.

```
LOC    OBJ      LINE     SOURCE

                   1
                   2
00000A             3     ALEN  EQU  10       ; define the array size
------             4     DSEC  SEGMENT IDATA ; define a IDATA segment
------             5     PSEC  SEGMENT CODE  ; define a CODE segment
                   6
------             7           RSEG DSEC     ; activate IDATA segment
000000             8     arr1: DS   ALEN     ; define arrays
00000A             9     arr2: DS   ALEN
                  10
------            11           RSEG PSEC     ; activate CODE segment
                  12     ; move memory block
                  13     ; %BMOVE (arr1,arr2,ALEN)
                  14     ;
                  15     ;      MOV  R0,#%src
                  16     ;      MOV  R1,#%dst
                  17     ;      MOV  R2,#%cnt
                  18     ; %lab: MOV  A,@R0
                  19     ;      MOV  @R1,A
                  20     ;      INC  R0
                  21     ;      INC  R1
                  22     ;      DJNZ R2, %lab
                  23
                  24     ;      MOV  R0,#%src
                  25     ;         arr1
000000 7E0000 F   26           MOV  R0,#arr1
                  27     ;      MOV  R1,#%dst
                  28     ;               arr2
000003 7E1000 F   29           MOV  R1,#arr2
                  30     ;      MOV  R2,#%cnt
                  31     ;               ALEN
000006 7E200A     32           MOV  R2,#ALEN
                  33     ; %lab: MOV  A,@R0
                  34     ;LAB0
000009 A5E6       35     LAB0: MOV  A,@R0
00000B A5F7       36           MOV  @R1,A
00000D A508       37           INC  R0
00000F A509       38           INC  R1
                  39     ;      DJNZ R2, %lab
                  40     ;               LAB0
000011 A5DA00 F   41           DJNZ R2, LAB0
                  42
                  43           END
```

The example shows an assembled file containing a macro definition in lines 1 to 9. The macro definition shows semicolons at start of each line. These semicolons are added by the assembler to prevent assembly of the definition text which is meaningful to the MPL preprocessor, but not to the remaining assembler phases. The listing will not include macro definitions or macro calls, if the general control **NOGEN** (which is the default if none is given), is used.

The macro BMOVE is called in line 12 with three actual parameters. Lines 14 to 20 shows the macro expansion, which is the return value of the macro call. This text will be assembled.

The example will produce assembly errors because no section directives are included in the source file. The purpose here is to show MPL processing, not the assembler semantics.

Local Symbols List

The DJNZ instruction in the previous example uses a local label for target of the branch. If a fixed label name is used (for example xlab, without leading %) then activation of the macro a second time would cause assembly errors due to multiple definitions of a single name.

The solution to this problem are local symbols. Local symbols are generated by the MPL processor as 'local_symbol_nnn', where local_symbol is the name of the local symbol and nnn is some number. Each time the macro is called, the number will be automatically incremented, the resulting names will be unique on each macro call.

The MPL processor increments a counter each time your program calls a macro that uses the LOCAL construct. The counter is incremented once for each symbol in the LOCAL list. Symbols in the LOCAL list, when used in the macro-body, receive a one to five digit suffix that is the decimal value of the counter. The first time you call a macro that uses the LOCAL construct, the suffix is '0'.

The syntax for the LOCAL construct in the DEFINE functions is shown below (this is finally the complete syntax for the MPL processor function DEFINE):

```
%*DEFINE (macro-name (parameter-list)) [LOCAL local-list] (macro-body)
```

The local-list is a list of valid macro identifiers separated by spaces or commas. The LOCAL construct in a macro has no affect on the syntax of a macro call.

Macro Processor Language Functions

The MPL processor has several predefined macro processor functions. These MPL processor functions perform many useful operations that would be difficult or impossible to produce in a user-defined macro. An important difference between a user-defined macro and a MPL processor function is that user-defined macros may be redefined, while MPL processor functions can not be redefined.

We have already seen one of these MPL processor functions, DEFINE. DEFINE creates user defined macros. MPL processor functions are already defined when the MPL processor is started.

Comment Function

The MPL processing language can be very subtle, and the operation of macros written in a straightforward manner may not be immediately obvious. Therefore, it is often necessary to comment macro definitions. The **comment function** has the following syntax:

```
%'text'
%'text end-of-line
```

The comment function always evaluates to the null string. Two terminating characters are recognized, the apostrophe and the end-of-line character. The second form allows to spread macro definitions over several lines, while avoiding unwanted end-of-lines in the return value. In either form of the comment function, the text or comment is not evaluated for macro calls.

Example

```
%'this is macro comment.'     ; this is an assembler comment.

%'the complete line including end-of-line is a comment
```

Source text before MPL processing

```
MOV    R5, R15    %'the following line will be kept separate'
MOV    R1,        %'this comment eats the newline character
R12
```

Output text from MPL processor

```
MOV    R5, R15
MOV    R1,        R12
```

Escape Function

Sometimes it is required to prevent the MPL processor from processing macro text. Two MPL processor functions perform this operation:

- escape function
- bracket function

The escape function interrupts scanning of macro text. The syntax of the **escape function** is:

```
%n text-n-characters-long
```

The metacharacter followed by a single decimal digit specifies the number of characters (maximum is 9) shall not be evaluated. The escape function is useful for inserting a metacharacter (normally the % character), a comma, or a parenthesis.

Example

```
10%1% OF 10 = 1;     expands to:     10% OF 10 = 1;
ASM%0251             expands to:     ASM251
```

Bracket Function

The other MPL processor function that inhibits the processing of macro text is the bracket function. The syntax of the **bracket function** is:

```
%(balanced-text)
```

The bracket function disables all MPL processing of the text contained within the parentheses. However, the escape function, the comment function, and parameter substitution are still recognized.

Since there is no restriction for the length of the text within the bracket function, it is usually easier to use than the escape function.

Example

```
ASM%(251)              evaluates to:      ASM251
%(1,2,3,4,5)           evaluates to:      1,2,3,4,5
```

Macro definition of 'DW'

```
%*DEFINE (DW (LIST, LABEL))  (
%LABEL:   DW     %LIST
)
```

Macro call to 'DW'

```
%DW (%(120, 121, 122, 123, -1), TABLE)
```

Return value of the macro call to 'DW'

```
TABLE:   DW     120, 121, 122, 123, -1
```

The macro above will add word definitions to the source file. It uses two parameters: one for the word expression list and one for the label name. Without the bracket function it would not be possible to pass more than one expression in the list, since the first comma would be interpreted as the delimiter separating the actual parameters to the macro. The bracket function used in the macro call prevents the expression list (120, 121, 122, 123, -1) from being evaluated as separate parameters.

METACHAR Function

The MPL processor function METACHAR allows the programmer to change the character that will be recognized by the MPL processor as the metacharacter. The use of this function requires extreme care.

The syntax of the **METACHAR** function is:

```
%METACHAR (balanced_text)
```

The first character of the balanced text is taken to be the new value of the metacharacter. The characters @, (,), *, blank, tab, and identifier-characters are not allowed to be the metacharacter.

Example

```
%METACHAR (!)          ; change metacharacter to '!'
!(1,2,3,4)             ; bracket function invoked with !
```

Numbers and Expressions

Balanced text strings appearing in certain places in built-in MPL processor functions are interpreted as numeric expressions:

- The argument to evaluate function **'EVAL'**
- The argument to the flow of control functions **'IF'**, **'WHILE'**, **'REPEAT'** and **'SUBSTR'**.

Expressions are processed as follows:

- The text of the numeric expression will be expanded in the ordinary manner of evaluating an argument to a macro function.
- The resulting string is evaluated to both a numeric and character representation of the expressions result. The return value is the character representation.

The following operators are allowed (shown in order of precedence).

1. Parenthesized Expressions
2. HIGH, LOW
3. *, /, MOD, SHL, SHR
4. EQ, LT, LE, GT, GE, NE
5. NOT
6. AND, OR, XOR

The arithmetic is done using signed 16-bit integers. The result of the relational operators is either 0 (FALSE) or 1 (TRUE).

Numbers

Numbers can be specified in hexadecimal (base 16), decimal (base 10), octal (base 8) and binary (base 2). A number without an explicit base is interpreted as

decimal, this being the default representation. The first character of a number must always be a digit between 0 and 9. Hexadecimal numbers which do not have a digit as the first character must have a 0 placed in front of them.

Base	Suffix	Valid Characters	Examples
hexadecimal	H,h	0 - 9, A-F (a - f)	1234H 99H 123H 0A0F0H 0FFH
		Hexadecimal numbers must be preceded with a 0, if the first digit is in range A to F	
decimal	D,d	0 - 9	1234 65590D 20d 123
octal	O,o,Q,q	0 - 7	177O 7777o 25O 123o 177777O
binary	B,b	0 - 1	1111B 10011111B 101010101B

Dollar ($) signs can be placed within the numbers to make them more readable. However a $ sign is not allowed to be the first or last character of a number and will not be interpreted.

1111$0000$1010$0011B is equivalent to **1111000010100011B**
1$2$3$4 is equivalent to **1234**

Hexadecimal numbers may be also entered using the convention from the C language:

```
0xFE02          0x1234
0X5566          0x0A
```

Character Strings

The MPL processor allows the use of ASCII characters strings in expressions. An expression is permitted to have a string consisting of one or two characters enclosed in single quote characters (').

'A'	evaluates to 0041H
'AB'	evaluates to 4142H
'a'	evaluates to 0061H
'ab'	evaluates to 6162H
''	the null string is not valid!
'abc'	ERROR due to more than two characters

The MPL processor cannot access the assembler's symbol table. The values of labels, SET and EQU symbols are not known during MPL processing. But, the programmer can define macro-time symbols with the MPL processor function 'SET'.

SET Function

The MPL processor function SET permits you to define macro-time symbols. SET takes two arguments: a valid identifier, and a numeric expression.

The syntax of the **SET** function is:

```
%SET (identifier,expression)
```

SET assigns the value of the numeric expression to the identifier.

The SET function affects the MPL processor symbol table only. Symbols defined by SET can be redefined with a second SET function call, or defined as a macro with DEFINE.

Source text

```
%SET (CNT, 3)
%SET (OFS, 16)
MOV R1,#%CNT+%OFS
%SET (OFS, %OFS + 10)
OFS = %OFS
```


Output text

```
MOV R1,#3+16
OFS = 26
```

The SET symbol may be used in the expression that defines its own value:

Source text

```
%SET (CNT, 10)                                    %' define variable CNT'
%SET (OFS, 20)                                    %' define variable OFS'
```

% 'change values for CNT and OFS'

```
%SET (CNT, %CNT+%OFS)                                          %' CNT = 30'
%SET (OFS, %OFS * 2)                                           %' OFS = 40'
MOV R2,#%CNT + %OFS                                                  %' 70'
MOV R5,#%CNT                                                         %' 30'
```

Output text

```
MOV R2,#30 + 40
MOV R5,#30
```

EVAL Function

The MPL processor function EVAL accepts an expression as an argument and returns the decimal character representation of it's result.

The syntax of the **EVAL** function is:

```
%EVAL (expression)
```

The expression arguments must be a legal expression with already defined macro identifiers, if any.

Source text

```
%SET (CNT, 10)                                    %' define variable CNT'
%SET (OFS, 20)                                    %' define variable OFS'

MOV  R15,#%EVAL (%CNT+1)
MOV  WR14,#%EVAL (14+15*200)
MOV  R13,#%EVAL (-(%CNT + %OFS - 1))
MOV  R2,#%EVAL (%OFS LE %CNT)
MOV  R7,#%EVAL (%OFS GE %CNT)
```

Output text

```
MOV  R15,#11
MOV  WR14,#3014
MOV  R13,#-29
MOV  R2,#0
MOV  R7,#1
```


Logical Expressions and String Comparison

The following MPL processor functions compare two balanced-text string arguments and return a logical value based on that comparison. If the function evaluates to 'TRUE,' then it returns '1'. If the function evaluates to 'FALSE,' then it returns '0'. The list of string comparison functions below shows the syntax and describes the type of comparison made for each. Both arguments to these function may contain macro calls. (These MPL calls are expanded before the comparison is made).

%EQS (arg1,arg2)	True if both arguments are identical
%NES (arg1,arg2)	True if arguments are different in any way
%LTS (arg1,arg2)	True if first argument has a lower value than second argument
%LES (arg1,arg2)	True if first argument has a lower value then second argument or if both arguments are identical
%GTS (arg1,arg2)	True if first argument has a higher value than second argument
%GES (arg1,arg2)	True if first argument has a higher value than second argument or if both arguments are identical

Example

```
%EQS (A251, A251)

LT%S (A251,a251)

%GTS (10,16)

%GES (a251,a251 )

%*DEFINE (VAR1) (A251)
%*DEFINE (VAR2) (%VAR1)
%EQS (%VAR1,%VAR2)
%EQS(A251,A251)
```

0 (FALSE), the space after the comma is part of the second argument

1 (TRUE), the lower case characters have a higher ASCII value than upper case

0 (FALSE), these macros compare strings not numerical values. ASCII '6' is greater than ASCII '1'

0 (FALSE), the space at the end of the second argument makes the second argument greater than the first

1 (TRUE) expands to:

6

Conditional MPL Processing

Some MPL functions accept logical expressions as arguments. The MPL uses the value 1 and 0 to determine TRUE or FALSE. If the value is one, then the expression is TRUE. If the value is zero, then the expression is FALSE.

Typically, you will use either the relational operators (EQ, NE, LE, LT, GT, or GE) or the string comparison functions (EQS, NES, LES, LTS, GTS, or GES) to specify a logical value.

IF Function

The IF MPL function evaluates a logical expression, and based on that expression, expands or skips its text arguments. The syntax of the MPL processor function **IF** is:

```
%IF (expression) THEN (balanced-text1) [ ELSE (balanced-text2) ] FI
```

IF first evaluates the expression, if it is TRUE, then balanced-text1 is expanded; if it is FALSE and the optional ELSE clause is included, then balanced-text2 is expanded. If it is FALSE and the ELSE clause is not included, the IF call returns a null string. FI must be included to terminate the call.

IF calls can be nested; when they are, the ELSE clause refers to the most recent IF call that is still open (not terminated by FI). FI terminates the most recent IF call that is still open.

Source text

```
%*DEFINE (ADDSUB (op,p1,p2))  (
 %IF (%EQS (%op,ADD)) THEN  (
  ADD    %p1,%p2
 )ELSE (%IF (%EQS (%op,SUB)) THEN (
   SUB   %p1,%p2
    ) FI
 ) FI
)

%ADDSUB (ADD,R15,R3)                                %' generate ADD R15,R3'
%ADDSUB (SUB,R15,R9)                                %' generate SUB R15,R9'
%ADDSUB (MUL,R15,R4)                                %' generates nothing !'
```

Output text

```
ADD    R15,R3
SUB    R15,R9
```

WHILE Function

Often you may wish to perform macro operations until a certain condition is met. The MPL processor function WHILE provides this facility.

The syntax for the MPL processor function **WHILE** is:

```
%WHILE (expression) (balanced-text)
```

WHILE first evaluates the expression. If it is TRUE, then the balanced-text is expanded; otherwise, it is not. Once the balanced-text has been expanded, the logical argument is retested and if it is still TRUE, then the balanced-text is again expanded. This loop continues until the logical argument proves FALSE.

Since the MPL continues processing until expression evaluates to FALSE, the balanced-text should modify the expression, or the WHILE may never terminate.

A call to the MPL processor function EXIT will always terminate a WHILE function. EXIT is described later.

Source text

```
%SET (count, 5)                                    %' initialize count to 5'
%WHILE (%count GT 0)
(   ADD   R15,R15 %SET (count, %count - 1)
)
```

Output text

```
ADD   R15,R15
ADD   R15,R15
ADD   R15,R15
ADD   R15,R15
ADD   R15,R15
```

REPEAT Function

The MPL processor function REPEAT expands its balanced-text a specified number of times. The syntax for the MPL processor function **REPEAT** is:

```
%REPEAT (expression) (balanced-test)
```

REPEAT uses the expression for a numerical value that specifies the number of times the balanced-text will be expanded. The expression is evaluated once when the macro is first called, then the specified number of iterations is performed.

Source text

```
%REPEAT (5)
( -enter any key to shut down-
)

%REPEAT (5) (+%REPEAT (9) (-))+
```

Output text

```
-enter any key to shut down-
-enter any key to shut down-
-enter any key to shut down-
-enter any key to shut down-
-enter any key to shut down-
```

EXIT Function

The EXIT MPL processor function terminates expansion of the most recently called REPEAT, WHILE or user-defined macro function. It is most commonly used to avoid infinite loops (example: a WHILE that never becomes FALSE, or a recursive user-defined macro that never terminates). It allows several exit points in the same macro.

The syntax for the MPL processor function **EXIT** is:

```
%EXIT
```

Source text

```
%SET (count, 0)

%WHILE (1)
(%IF (%count GT 5) THEN (%EXIT)
FI    DW   %count, -%count
%SET (count, %count + 1))
```

Output text

```
DW  0, -0
DW  1, -1
DW  2, -2
DW  3, -3
DW  4, -4
DW  5, -5
```


String Manipulation Functions

The purpose of the MPL is to manipulate character strings. Therefore, there are several MPL functions that perform common character string manipulations.

LEN Function

The MPL processor function LEN returns the length of the character string argument in hexadecimal: The character string is limited to 256 characters.

The syntax for the MPL processor function **LEN** is:

```
%LEN (balanced-text)
```

Source text

```
%LEN (A251)                                                    %' len = 4'
%LEN (A251,A251)                                        %' comma counts also'
%LEN ()
%LEN (ABCDEFGHIJKLMNOPQRSTUVWXYZ)
%DEFINE (TEXT) (QUEEN)
%DEFINE (LENGTH) (%LEN (%TEXT))
LENGTH OF '%TEXT' = %LENGTH.
```

Output text

```
4
9
0
26
LENGTH OF 'QUEEN' = 5.
```

SUBSTR Function

The MPL processor function SUBSTR returns a substring of the given text argument. The function takes three arguments: a character string to be divided and two numeric arguments.

The syntax for the MPL processor function **SUBSTR** is:

```
%SUBSTR (balanced-text,expression1,expression2)
```

balanced-text is any text argument, possibly containing macro calls. Expression1 specifies the starting character of the substring. Expression2 specifies the number of characters to be included in the substring.

If expression1 is zero or greater than the length of the argument string, then SUBSTR returns the null string. The index of the first character of the balanced text is one.

If expression2 is zero, then SUBSTR returns the null string. If expression2 is greater than the remaining length or the string, then all characters from the start character to the end of the string are included.

Source text

```
%DEFINE (STRING) (abcdefgh)
%SUBSTR (%string, 1, 2)
%SUBSTR (%(1,2,3,4,5), 3, 20)
```

Output text

```
ab
2,3,4,5
```

MATCH Function

The MPL processor function MATCH searches a character string for a delimiter character, and assigns the substrings on either side of the delimiter to the identifiers.

The syntax for the MPL processor function **MATCH** is:

```
%MATCH (identifier1 delimiter identifier2) (balanced-text)
```

Identifier1 and identifier2 must be valid macro identifiers. Delimiter is the first character to follow identifier1. Typically, a space or comma is used, but any character that is not a macro identifier character may be used. Balanced-text is the text searched by the MATCH function. It may contain macro calls.

MATCH searches the balanced-text string for the specified delimiter. When the delimiter is found, then all characters to the left are assigned to identifier1 and all characters to the right are assigned to identifier2. If the delimiter is not found, the entire balanced-text string is assigned to identifier1 and the null string is assigned to identifier2.

Source text

```
%DEFINE (text) (-1,-2,-3,-4,-5)
%MATCH (next,list) (%text)
%WHILE (%LEN (%next) NE 0)
(       MOV    R8,#%next
        MOV    @WR2,R8 %MATCH (next,list)(%list)
        INC    WR2,#1
)
```

Output text

```
MOV    R8,#-1
MOV    @WR2,R8
INC    WR2,#1
MOV    R8,#-2
MOV    @WR2,R8
INC    WR2,#1
MOV    R8,#-3
MOV    @WR2,R8
INC    WR2,#1
MOV    R8,#-4
MOV    @WR2,R8
INC    WR2,#1
MOV    R8,#-5
MOV    @WR2,R8
INC    WR2,#1
```

Console I/O Functions

There are two MPL processor functions that perform console I/O: IN and OUT. Their names describe the function each performs. IN outputs a character '>' as a prompt, and returns the line typed at the console. OUT outputs a string to the console; a call to OUT is replaced by the null string.

The syntax for the MPL processor functions **IN** and **OUT** is:

```
%IN
%OUT (balanced-text)
```


Source text

```
%OUT (enter baud rate)
%set (BAUD_RATE,%in)
BAUD_RATE = %BAUD_RATE
```

Output text

```
<19200 was entered at the console>
BAUD_RATE = 19200
```

Advanced Macro Processing

The MPL definition function associates an identifier with a functional string. The macro may or may not have an associated pattern consisting of parameters and/or delimiters. Also optionally present are local symbols. The syntax for a **macro definition** is:

```
%DEFINE (macro_id define_pattern) [LOCAL id_list] (balanced_text)
```

The define_pattern is a balanced string which is further analyzed by the MPL processor as follows:

```
define_pattern = { [parm_id] [delimiter_specifier] }
```

Delimiter_specifier is one of the following:

- some string not containing non-literal id-continuation. logical blank or character @.
- @delimiter_id

The macro call must have a call pattern which corresponds to the macro define pattern. Regardless of the type of delimiter used to define a macro, once it has been defined, only delimiters used in the definition can be used in the macro call. Macros defined with parentheses and commas require parentheses and commas in the macro call. Macros defined with spaces or any other delimiter require that delimiter when called.

The define pattern may have three kinds of delimiters: implied blank delimiters, identifier delimiters and literal delimiters.

Literal Delimiters

The delimiters used in user-defined macros (parentheses and commas) are literal delimiters. A literal delimiter can be any character except the metacharacter.

When you define a macro using a literal delimiter, you must use exactly that delimiter when you call the macro. If the specified delimiter is not used as it appears in the definition, a macro error occurs.

When defining a macro, the delimiter string must be literalized, if the delimiter meets any of the following conditions:

- more than one character,
- a macro identifier character (A-Z, 0-9, _, or ?),
- a commercial at (@), a space, tab, carriage return, or linefeed.

Use the escape function (%n) or the bracket function (%()) to literalize the delimiter string.

This is the simple form shown earlier:

Before Macro Expansion	After Macro Expansion
%*DEFINE(MAC(A,B))(%A %B)	null string
%MAC(4,5)	4 5

In the following example brackets are used instead of parentheses. The commercial at symbol separates parameters:

```
%*DEFINE (MOV[A%(@)B]) (MOV %A,%B)      →     null string
%MOV[P0@P1]                             →      MOV P0,P1
```

In the next two examples, delimiters that could be id delimiters have been defined as literal delimiter (the differences are noted):

```
%*DEFINE(ADD (R10 AND B)) (ADD R10,%B)      →     null string
%ADD (R10 AND #27H)                         →     ADD R10,#27H
```

Spaces around AND are considered as part of the argument string.

Blank Delimiters

Blank delimiters are the easiest to use. Blank delimiter is one or more spaces, tabs or new lines (a carriage-return/linefeed pair) in any order. To define a macro that uses the blank delimiter, simply place one or more spaces, tabs, or new lines surrounding the parameter list.

When the macro defined with the blank delimiter is called, each delimiter will match a series of spaces, tabs, or new lines. Each parameter in the call begins with the first non-blank character, and ends when a blank character is found.

Source text

```
%*DEFINE (X1 X2 X3) (P2=%X2, P3=%X3)
%X1 assembler A251
```

Output text

```
P2=assembler, P3=A251
```

Identifier Delimiters

Identifier delimiters are legal macro identifiers designated as delimiters. To define a macro that uses an identifier delimiter, you must prefix the delimiter with the @ symbol. You must separate the identifier delimiter from the macro identifiers (formal parameters or macro name) by a blank character.

When calling a macro defined with identifier delimiters, a blank delimiter is required to precede the identifier delimiter, but none is required to follow the identifier delimiter.

Source text

```
%*DEFINE (ADD X1 @TO X2 @STORE X3)(
  MOV    R1,%X1
  MOV    R2,%X2
  ADD    R1,R2
  MOV    %X3,R1
)

%ADD VAR1 TO VAR2 STORE VAR3
```

Output text

```
MOV    R1,VAR1
MOV    R2,VAR2
ADD    R1,R2
MOV    VAR3,R1
```

Literal and Normal Mode

In normal mode, the MPL processor scans for the metacharacter. If it is found, parameters are substituted and macros are expanded. This is the usual operation of the MPL processor.

When the literal character (*) is placed in a DEFINE function, the MPL processor shifts to literal mode while expanding the macro. The effect is similar

to surrounding the entire call with the bracket function. Parameters to the literalized call are expanded, the escape, comment, and bracket functions are also expanded, but no further processing is performed. If there are any calls to other macros, they are not expanded.

If there are no parameters in the macro being defined, the DEFINE function can be called without literal character. If the macro uses parameters, the MPL processor will attempt to evaluate the formal parameters in the macro-body as parameterless macro calls.

The following example illustrates the difference between defining a macro in literal mode and normal mode:

```
%SET (TOM, 1)
%*DEFINE (AB) (%EVAL (%TOM))
%DEFINE (CD) (%EVAL (%TOM))
```

When AB and CD are defined, TOM is equal to 1. The macro body of AB has not been evaluated due to the literal character, but the macro body of CD has been completely evaluated, since the literal character is not used in the definition. Changing the value of TOM has no effect on CD, but it changes the value of AB:

```
%SET (TOM,2)          → null string
%AB                   → 2
%CD                   → 1
%*CD                  → 1
%*AB                  → %EVAL (%TOM)
```

MACRO Errors

The MPL processor will output error messages, if errors occur in the MPL processing phase. The errors are displayed like other assembly errors in the listing file which is generated by the assembler anyway. The following describes the error messages generated by the MPL processor.

Number	Error Message and Description
200	**PREMATURE END OF FILE** The end of the source module was reached while processing some macro call, which requires more input from the source file.
201	**'<token>' IDENTIFIER EXPECTED** The MPL processor expected an identifier while processing some macro. None was found. The unexpected token is displayed with this error message.
202	**MPL FUNCTION '<name>': '<character>' EXPECTED** The context of the MPL processor language requires a specific character from the input given by <character> while processing the built-in function given by <name>.
203	**<string>: UNBALANCED PARENTHESES** A balanced string requires the same number of right parentheses and left parentheses.
204	**EXPECTED '<token>'** The syntax requires a specific token to follow, for example THEN after the balanced text argument to IF.
205	**INCOMPLETE MACRO DEFINITION** The macro definition has not been completely processed due to premature end of input file.
206	**FUNCTION 'MATCH': ILLEGAL CALL PATTERN** The built-in function MATCH was called with an illegal call pattern. The call pattern must consist of some formal name followed by a delimiter specification and another formal name.
207	**FUNCTION 'EXIT' IN BAD CONTEXT** The built-in function EXIT is allowed only in the loop control constructs WHILE and REPEAT.
208	**ILLEGAL METACHARACTER '<character>'** The first character of the balanced text argument to METACHAR is taken to be the new value of the metacharacter. The characters @, (,), *, blank, tab, and identifier-characters are not allowed to be the metacharacter.
209	**CALL PATTERN - DELIMITER '<delimiter>' NOT FOUND** The call pattern of some macro does not conform to the define pattern of that macro. The delimiters of the macro call should be checked for conformance.
210	**CALL TO UNDEFINED MACRO '<name>'** The macro call specifies the name of an undefined macro.

Number	Error Message and Description
211	**INVALID MPL COMMAND '%<character>'** The character following the metacharacter does not form a valid MPL command.
212	**INVALID DIGIT '<character>' IN NUMBER** A number of an expression contains an invalid digit.
213	**UNCLOSED STRING OR CHARACTER CONSTANT**
214	**INVALID STRING OR CHARACTER CONSTANT** The string representing a number in an expression is invalid. The string must be either one or two characters long. A character constant must not be longer than one character. Strings or character constants must be enclosed by single or double quotes.
215	**UNKNOWN EXPRESSION IDENTIFIER** The identifier within some expression is not an operator or a number.
216	**<character>: INVALID EXPRESSION TOKEN** The given character does not form a valid operator or an identifier operator.
217	**DIV/MOD BY ZERO** A division or modulo by zero error occurred while evaluating an expression.
218	**EVAL: SYNTAX ERROR IN EXPRESSION** The expression to be evaluated contains a syntax error, for example two consecutive number, not separated by an operator.
219	**CAN'T OPEN FILE '<file>'** The file specified in the INCLUDE directive could not be opened.
220	**'<file>' IS NOT A DISK FILE** The file name given in the INCLUDE directive does not specify a disk file. Files other than disk files are not allowed (example: CON).
221	**ERROR IN INCLUDE DIRECTIVE** The INCLUDE directive is ill-formed. The argument to INCLUDE must be the name of some file, enclosed in parentheses.

Chapter 7. Invocation and Controls

This part explains how to use A251 to assemble 8051 assembly source files and discusses the assembler controls that may be specified on the command line and within the source file.

Using the controls described in this part, you can specify which operations are performed by A251. For example, you can direct A251 to generate a listing file, produce cross reference information, and control the amount of information included in the object file. You can also conditionally assemble sections of code using the conditional assembly controls.

Running A251

The A251 assembler is invoked by typing A251 at the DOS prompt. The command line must contain the name of an 8051 assembly source file to be assembled as well as any command–line controls that are required. The format for the A251 command line is:

```
A251 sourcefile [ controls… ]
```

where

sourcefile is the name of the source program you want to assemble. A251assembler

controls are used to direct the operation of the assembler. Refer to “Assembler Controls” on page 115 for more information.

The following command line example invokes A251 and specifies the source file **SAMPLE.A51** and uses the controls **DEBUG**, **XREF**, and **PAGEWIDTH**.

```
A251 SAMPLE.A51 DEBUG XREF PAGEWIDTH(132)
```

A251 displays the following information upon successful invocation and assembly.

```
DOS MACRO ASSEMBLER A251 V1.00

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Command Files

Command files are ASCII text files that contain information that you would normally type on the DOS invocation line. Command files can include the name of the source file to assemble as well as any assembler controls.

A251 allows you to specify a command file on the DOS invocation line using an at sign (**@**).

Example

```
A251 @CMDFIL
```

The contents of the file **CMDFIL** will be interpreted as one long input command line.

DOS ERRORLEVEL

After assembly, the number of errors and warnings detected is output to the screen. A251 then sets the DOS **ERRORLEVEL** to indicate the status of the assembly. Values are listed in the following table:

ERROR LEVEL	Meaning
0	No ERRORS or WARNINGS
1	WARNINGS only
2	ERRORS and possibly also WARNINGS
3	FATAL ERRORS

You can access the **ERRORLEVEL** variable in DOS batch files for conditional inquiries in order to terminate the batch processing when an error occurs. Refer to your DOS User's Guide for more information about **ERRORLEVEL** or batch files.

Output Files

A251 generates a number of output files during assembly. By default, each of these shares the same *basename* as the source file. However, each has a different file extension. The following table lists the files and gives a brief description of each.

File Extension	Description
***basename*.LST**	Files with this extension are listing files that contain the formatted source text along with any errors detected by the assembler. Listing files may optionally contain symbols used and the generated assembly code. Refer to “PRINT / NOPRINT” on page 141 for more information.
***basename*.OBJ**	Files with this extension are object modules that contain relocatable object code. Object modules can be linked into an absolute object module by the L51 Linker/Locator. Refer to “OBJECT / NOOBJECT” on page 138 for more information.

Assembler Controls

A251 provides a number of controls that you can use to direct the operation of the assembler. Controls are composed of one or more letters or digits and, unless otherwise indicated, can be specified after the filename on the invocation line or in a control line within the source file. Control lines are prefixed by the dollar sign (**$**).

Example

```
A251 TESTFILE.A51 MPL DEBUG XREF
```

or

```
$MPL
$DEBUG
$XREF
```

or

```
$MPL DEBUG XREF
```

In the above example, **MPL**, **DEBUG**, and **XREF** are all control commands and **TESTFILE.A51** is the source file that will be assembled.

A251 has two classes of controls: primary and general. The primary controls are set in the invocation line or the primary control lines and remain in effect throughout the assembly. For this reason, primary controls may be used only in the invocation line or in the control line at the beginning of the program. Only other control lines (that do not contain the INCLUDE control) may precede a line containing a primary control. The INCLUDE control terminates of primary controls.

If a primary control is specified in the invocation line and in the primary control lines, the first time counts. This enables the programmer to override primary controls via the invocation line.

The general controls are used to control the immediate action of the assembler. Typically their status is set and modified during the assembly. Control lines containing only general controls may be placed anywhere in your source code.

The table below lists all of the controls, their abbreviations, their default values, and a brief description of each.

Name / Abbreviation	Meaning
DATA(*date*) / DA	Places a date string in header (9 characters maximum).
CASE	Enable case sensitive mode for symbol names.
DEBUG / DB	Outputs debug symbol information to object file.
EJECT / EJ ‡	Continue listing on next page.
ERRORPRINT[(*file*)] / EP	Designates a file to receive error messages in addition to the listing.
GEN / GE ‡	Generates a full listing of macro expansions in the listing file.
NOGEN / NOGE ‡	List only the original source text in listing file.
INCLUDE(*file*) / IC ‡	Designates a file to be included as part of the program.
LINK ‡	Place Linker/Locator controls in the Assembler source code.
LIST, NOLIST / LI, NOLI ‡	Print or do not print the assembler source in the listing file.
MODBIN / MB	Select MCS 251 binary mode (default).
MODSRC / MS	Select MCS 251 source mode.
MPL	Enable Macro Processing Language.
NOAMAKE	Disable AutoMAKE information.
NOLINES	Do not generate LINE number information.
NOMACRO / NOMR	Disable Standard Macros
NOMOD51 / NOMO	Do not recognize the 8051-specific predefined special register.
NOMOD251 / NO251	Disable the additional MCS 251 instructions.
NOOBLECT / NOOJ	Designates that no object file will be created.
NOREGISTERBANK/ NORB	Indicates that no banks are used.
NOSYMBOLS / NOSB	No symbol table is listed.
NOSYMLIST,NO SL ‡	Do not list the following symbol definitions in the symbol table.
OBJECT[(*file*)] / OJ	Designate file to receive object code.
PAGELENGTH(*n*) / PL	Sets maximum number of lines in each page of listing file.
PAGEWIDTH(n) / PW	Sets maximum number of characters in each line of listing file.
PRINT[(*file*)] / PR	Designates file to receive source listing.
NOPRINT / NOPR	Designates that no listing file will be created.

Name / Abbreviation	Meaning
REGISTERBANK(*num,...*)	Indicates one or more banks used in program module.
REGUSE ‡	Defines register usage of assembler functions for the C optimizer.
RESTORE / RS ‡	Restores control setting from SAVE stack.
SAVE / SA ‡	Stores current control setting for GEN, LIST and SYMLIST.
SYMLIST, SL ‡	List the following symbol definitions in the symbol table.
TITLE(*string*) / TT	Places a string in all subsequent page headers.
XREF / XR	Creates a cross reference listing of all symbols used in program.

‡ — General controls

NOTE

*Some controls like **EJECT** and **SAVE** cannot be specified on the command line. The syntax for each control is the same when specified on the command line or when specified within the source file. A251 will generate a fatal error for controls that are improperly specified.*

COND / NOCOND

Abbreviation: None.

Arguments: None.

Default: **COND**

Control Class: General

Description: The **COND** control directs A251 to include unassembled parts of a conditional **IF**–**ELSEIF**–**ENDIF** construct in the listing file. Unassembled code is listed without line numbers.

The **NOCOND** control prevents unassembled portions of an **IF**–**ELSE**–**ENDIF** block from appearing in the listing file.

Examples:

```
A251 SAMPLE.A51 COND

$COND

A251 SAMPLE.A51 NOCOND

$NOCOND
```

DATE

Abbreviation: **DA**

Arguments: A string enclosed within parentheses.

Default: The date obtained from the operating system.

Control Class: Primary

Description: A251 includes the current date in the header of each page in the listing file. The **DATE** control allows you to specify the date string that is included in the header. The string must immediately follow the **DATE** control and must be enclosed within parentheses. Only the first 8 characters of the date string are used. Additional characters are ignored.

Example:

```
A251 SAMPLE.A51 DATE(19JAN92)

$DATE(10/28/91)
```

7

CASE †

Abbreviation: **CA**

Arguments: None.

Default: No case sensitivity.

Control Class: Primary

Description: The assembler is directed to operate in case sensitive mode (**CASE**) or case insensitive mode. In case insensitive mode the assembler maps lower case input characters to upper case. **CASE** becomes meaningful if modules generated by the assembler are combined with modules generated from the C compiler. Identifiers exported from C modules appear always as written, the corresponding names in the assembler module must therefore put into the object module as written, preserving case sensitivity.

Example:

```
$CASE

A251 SAMPLE.A51 CASE
```

DEBUG

Abbreviation: **DB**

Arguments: None.

Default: No debugging information is generated.

Control Class: Primary

Description: The **DEBUG** control instructs A251 to include debugging information in the object file. This information is used when testing the program with an emulator or simulator.

The **DEBUG** control also includes line number information for source level debugging. This line number information can be disabled with the **NOLINES** control.

Examples:

```
A251 SAMPLE.A51 DEBUG

$DEBUG
```

EJECT

Abbreviation: **EJ**

Arguments: None

Default: None

Control Class: General

Description: The **EJECT** control inserts a form feed into the listing file after the line containing the **EJECT** control statement. This control is ignored if **NOLIST** or **NOPRINT** was previously specified.

Example:

```
$EJECT
```

ERRORPRINT

Abbreviation: **EP**

Arguments: An optional filename enclosed within parentheses

Default: No error messages are output to the console.

Control Class: Primary

Description: The **ERRORPRINT** control directs A251 to output all error messages either to the console or to a specified file. If no filename is specified with the **ERRORPRINT** control, all error messages are output to the console.

Examples:

```
A251 SAMPLE.A51 ERRORPRINT(SAMPLE.ERR)

A251 SAMPLE2.A51 ERRORPRINT

$EP
```

GEN / NOGEN

Abbreviation: **GE / NOGE**

Arguments: None

Default: **NOGEN**

Control Class: General

Description: The **GEN** control directs A251 to expand or list all assembly instructions contained in a macro.

The **NOGEN** control prevents the A251 assembler from including macro expansion text in the listing file. Only the macro name is listed.

Examples:

```
A251 SAMPLE.A51 GEN

$GEN

A251 SAMPLE.A51 NOGEN

$NOGEN
```

INCLUDE

Abbreviation: **IC**

Arguments: A filename enclosed within parentheses.

Default: None.

Control Class: General

Description: The **INCLUDE** control directs A251 to include the contents of the specified file in the assembly of the program immediately following the control line. **INCLUDE** files may be nested up to 9 deep.

The **INCLUDE** control is usually used to include special function register definition files for 8051 and MCS 251 derivatives as well as to include declarations for external routines, variables, and macros. Files containing assembly language code may also be included.

Example:

```
$INCLUDE (REG252.DCL)
```

LINK †

Abbreviation: **LI**

Arguments: Linker/Locator control directives enclosed in parentheses.

Default: None

Control Class: General

Description: The **LINK** control allows you to include Linker/Locator control directives into the assembler source. The control directives specified within the assembler source will be pass to the Linker/Locator as they would be specified in the invocation line of the linker/locator. The **LINK** control is useful to correct directly in the assembler source the overlay analysis of your application, if your program contains indirect function calls.

For more information about Linker/Locator controls refer to the *8051 Utilities User's Guide*.

The **LINK** control cannot be specified in the A251 invocation line.

Example:

```
$LINK (ADDCALL (MYFUNC ! MYFUNC2)
```

LIST / NOLIST

Abbreviation: **LI / NOLI**

Arguments: None

Default: **LIST**

Control Class: General

Description: The **LIST** control directs A251 to include the program source text in the generated listing file.

The **NOLIST** control prevents subsequent lines of your assembly program from appearing in the generated listing file. If a line that would normally not be listed causes an assembler error, that line will be listed along with the error message.

Examples:

```
A251 SAMPLE.A51 LI

$LIST

A251 SAMPLE.A51 NOLIST

$NOLI
```

MACRO / NOMACRO

Abbreviation: **NOMR**

Arguments: None

Default: **MACRO**

Control Class: Primary

Description: The **MACRO** control instructs the A251 assembler to recognize and process macro definitions and invocations.

The **NOMACRO** control disables the macro processor in A251. Macros will not be expanded.

Examples:

```
A251 SAMPLE.A51 NOMACRO

$NOMACRO
```

MODBIN †

Abbreviation: **MB**

Arguments: None

Default: **MODBIN**

Control Class: Primary

Description: The **MODBIN** control instructs the A251 assembler to generate code for the MCS 251 architecture using the BINARY mode of this CPU.

See also: **MODSRC, NOMOD251**

Examples:

```
A251 SAMPLE.A51 MODBIN

$MODBIN
```

MODSRC †

Abbreviation: **MS**

Arguments: None

Default: **MODBIN**

Control Class: Primary

Description: The **MODSRC** control instructs the A251 assembler to generate code for the MCS 251 architecture using the SOURCE mode of this CPU.

See also: **MODBIN, NOMOD251**

Examples:

```
A251 SAMPLE.A51 MODSRC

$MODSRC
```

MPL

Abbreviation: None

Arguments: None

Default: The Macro Processing Language is disabled.

Control Class: Primary.

Description: The **MPL** control enables the Macro Processing Language. The **MPL** language is compatible to the Intel ASM51. Refer to “Chapter 6. Macro Processing Language” on page 87 for more information about the MPL processor.

Examples:

```
A251 SAMPLE.A51 MPL

$MPL
```

NOAMAKE

Abbreviation: **NOAM**

Arguments: None.

Default: Generate AutoMAKE information.

Control Class: Primary

Description: **NOAMAKE** disables the project information records of the A251 Macro Assembler for use with the automatic MAKE utility AutoMAKE. This option disables also the register information given with the REGUSE directive. If NOAMAKE is used, the generated object files can be used with older program versions of the 8051 development tool chain.

Example:

```
A251 SAMPLE.A51 NOAMAKE

$ NOAMAKE
```

NOLINES

Abbreviation: **NOLI**

Arguments: None.

Default: Line numbers for source level debugging are generated when the **DEBUG** control is used.

Control Class: Primary

Description: The **NOLINES** control disables the line number information for source level debugging. This control is useful when A251 should be used in connection with old debugging tools or emulators.

Examples:

```
A251 SAMPLE.A51 NOLINES

$NOLINES
```

NOMACRO

Abbreviation: None.

Arguments: None.

Default: Standard Macros are fully expanded.

Control Class: Primary

Description: The **NOMACRO** control disables the standard macro facility of A251. Standard macros are not expanded.

Examples:

```
A251 SAMPLE.A51 NOMACRO

$NOMACRO
```

NOMOD51

Abbreviation: **NOMO**

Arguments: None.

Default: In A51 all special function registers of the 8051 CPU are predefined. A251 does not define CPU special function registers at all.

Control Class: Primary

Description: The **NOMOD51** control prevents the A51 assembler from implicitly defining symbols for the default 8051 special function registers. This is necessary when you want to include a definition file to declare symbols for the special function registers of a different 8051 derivative.

The A251 assembler supports the **NOMOD51** control only for source compatibility to the A51. However the 8051 special function registers are not predefined in A251.

Examples:

```
A251 SAMPLE.A51 NOMO

$NOMOD51
```

NOMOD251 †

Abbreviation: **NO251**

Arguments: None.

Default: Support the additional MCS 251 instructions.

Control Class: Primary

Description: The **NOMOD251** control disables the enhance MCS 251 instruction set. Only the original 8051 instructions are supported. With this control the A251 can be used to generate code for the 8051 architecture only.

See also: **MODBIN, MODSRC**

Examples:

```
A251 SAMPLE.A51 NO251

$NOMOD251
```

NOSYMBOLS

Abbreviation: **SB / NOSB**

Arguments: None

Default: A251 generates a table of all symbols used in and by the assembly program module. This symbol table is included in the generated listing file.

Control Class: Primary

Description: The **NOSYMBOLS** control prevents A251 from generating a symbol table in the listing file.

Examples:

```
A251 SAMPLE.A51 SYMBOLS

$SB

A251 SAMPLE.A51 NOSB

$NOSYMBOLS
```

OBJECT / NOOBJECT

Abbreviation: **OJ / NOOJ**

Arguments: An optional filename enclosed within parentheses.

Default: **OBJECT (*basename*.OBJ)**

Control Class: Primary

Description: The **OBJECT** control specifies that the A251 assembler generate an object file. The default name for the object file is ***basename*.OBJ**, however, an alternate filename may be specified in parentheses immediately following the **OBJECT** control statement.

The **NOOBJECT** control prevents A251 from generating an object file.

Examples:

```
A251 SAMPLE.A51 OBJECT (OBJDIR\SAMPLE.OBJ)

OJ(OBJ\SAMPLE.OBJ)

A251 SAMPLE.A51 NOOJ

$NOOBJECT
```

7

PAGELENGTH

Abbreviation: **PL**

Arguments: A number between 10 and 65535 enclosed within parentheses.

Default: **PAGELENGTH (60)**

Description: The **PAGELENGTH** control specifies the number of lines printed per page in the listing file. The number must be a decimal value between 10 and 65535. The default is 60.

Example:

```
A251 SAMPLE.A51 PAGELENGTH(132)

$PL (66)
```

PAGEWIDTH

Abbreviation: **PW**

Arguments: A number between 78 and 132 enclosed within parentheses.

Default: **PAGEWIDTH (120)**

Control Class: Primary

Description: The **PAGEWIDTH** control specifies the maximum number of characters in a line in the listing file. Lines that are longer than the specified width are automatically wrapped around to the next line. The default number of characters per line is 120.

Example:

```
A251 SAMPLE.A51 PW(79)

$PW(132)
```

PRINT / NOPRINT

Abbreviation: **PR / NOPR**

Arguments: An optional filename enclosed within parentheses.

Default: **PRINT(*basename*.LST)**

Control Class: Primary

Description: The **PRINT** control directs the A251 assembler to generate a listing file. The default name for the listing file is ***basename*.LST**, however, an alternate filename may be specified in parentheses immediately following the **PRINT** control statement.

The **NOPRINT** control prevents A251 from generating a listing file.

Examples:

```
A251 SAMPLE.A51 PRINT

A251TESTPRG.A51 PR(TESTPRG1.LST)

$PRINT(LPT1)

A251 SAMPLE.A51 NOPRINT

$NOPR
```

REGISTERBANK / NOREGISTERBANK

Abbreviation: **RB / NORB**

Arguments: Register bank numbers separated by commas and enclosed within parentheses; e.g., **REGISTERBANK (1,2,3)**.

Default: **REGISTERBANK (0)**

Control Class: Primary

Description: The **REGISTERBANK** control specifies the register banks used in the source module. This information is stored in the generated object file for further processing by the L251 Linker/Locator.

The **NOREGISTERBANK** control specifies that A251 reserves no memory for the register bank.

Examples:

```
A251 RBUSER.A51 REGISTERBANK(0,1,2)

$RB(0,3)

A251 SAMPLE.A51 NOREGISTERBANK

$NORB
```

REGUSE

Abbreviation: **RU**

Arguments: Name of a PUBLIC symbol and a register list enclosed in parentheses.

Default: Not applicable.

Control Class: General

Description: The **REGUSE** control specifies the registers modified during a function execution. The REGUSE control can be used in combination with the C51 or C251 C compiler and allows the global register optimization also for functions written in assembler language. For more information about global register optimization refer to the *C51 Compiler User's Guide* or the *C251 Compiler User's Guide*.

The **REGUSE** cannot be specified on the A251 invocation line.

Examples:

```
$REGUSE MYFUNC (ACC, B, R0 - R7)

$REGUSE PROCA (DPL, DPH)

$REGUSE PUTCHAR (R6,R7, CY, ACC)
```

RESTORE

Abbreviation: **RS**

Arguments: None.

Default: None.

Control Class: General

Description: The **RESTORE** control fetches and restores the values of the **GEN** and **LIST** controls that were stored by the last **SAVE** control statement.

See Also: **SAVE**

Example:

```
.
.
.
$SAVE
$NOLIST
.
.
.
$RESTORE
.
.
.
```

SAVE

Abbreviation: **SA**

Arguments: None

Default: None

Control Class: General

Description: The **SAVE** control stores the current settings of the **LIST** and **GEN** controls. Subsequent controls can modify the **LIST** and **GEN** settings.

This control allows these settings to be saved, altered for a number of program lines, and restored using the **RESTORE** control. The **SAVE** control can be nested up to nine times.

See Also: **RESTORE**

Example:

```
.
.
.
$SAVE
$INCLUDE(SAMPLE.INC)
$RESTORE
.
.
.
```

SYMLIST / NOSYMLIST

Abbreviation: **SL/NOSL**

Arguments: None.

Default: **SYMLIST**

Control Class: General

Description: The **SYMLIST** control enables the listing of symbol definitions in the symbol table.

The **NOSYMLIST** control prevents the A251 assembler from listing of symbol definitions in the symbol table. The **NOSYMLIST** control is useful in special function register definition files.

Examples:

```
A251 SAMPLE.A51 NOMO

$NOMOD51
```

TITLE

Abbreviation: **TT**

Arguments: A string enclosed within parentheses.

Default: The *basename* of the source file excluding the extension.

Control Class: General

Description: The **TITLE** control allows you to specify the title to use in the header line of the listing file. The text to used for the title must immediately follow the **TITLE** control and must be enclosed in parentheses. A maximum of 60 characters may be specified for the title.

Example:

```
A251 SAMPLE.A51 TITLE(Oven Controller Version 3)

$TT(Race Car Controller)
```

XREF

Abbreviation: **XR**

Arguments: None.

Default: No error references are listed.

Description: The **XREF** control directs the A251 assembler to generate a cross reference table of the symbols used in the source module. The alphabetized cross reference table will be included in the generated listing file.

Example:

```
A251 SAMPLE.A51 XREF

$XREF
```

Directives for Conditional Assembly

The directives for conditional assembly belong to the class of general controls. Conditional assembly can be used to implement different program versions of different memory models with one source file. Therefore only one source module must be maintained to satisfy several applications

Text blocks to be conditionally assembled are enclosed by **IF**, **ELSEIF**, **ELSE** and **ENDIF**.

The **SET** and **RESET** directives may be used in the invocation line of the assembler. The remaining instructions for conditional assembly are only allowed within the source file and cannot be part of the assembler invocation line.

IF blocks may be nested to a maximum of ten. If a block is not translated the nested conditional blocks which are part of this block also skipped.

Conditional Assembly Controls

Conditional assembly controls allow you to write 8051 assembly programs with sections that can be included or excluded from the assembly based on the value of a constant expression. Blocks that are to be conditionally assembled are enclosed by **IF**, **ELSEIF**, **ELSE**, and **ENDIF** control statements.

The conditional control statements **IF**, **ELSE**, **ELSEIF**, and **ENDIF** can be specified only in the source program. They are not allowed on the invocation line. Additionally, these controls can be specified both with and without the leading dollar sign (**$**).

When prefixed with a dollar sign, the conditional control statements can only access symbols defined by the **SET** and **RESET** controls.

When specified without a dollar sign, the conditional control statements can access all symbols except those defined by the **SET** and **RESET** controls. These control statements can access parameters in a macro definition.

IF blocks may be nested up to 10 levels deep, however, if an **IF**, **ELSEIF**, or **ELSE** block is not assembled, any **IF** blocks nested therein are also not assembled.

The following table lists the conditional assembly control statements.

Control	Meaning
IF	Translate block if condition is true
ELSE	Translate block if the condition of a previous **IF** is false.
ELSEIF	Translate block if condition is true and a previous **IF** or **ELSEIF** is false.
ENDIF	Marks end of a block.
RESET	Set symbols checked by **IF** or **ELSEIF** to false.
SET	Set symbols checked by **IF** or **ELSEIF** to true or to a specified value.

SET

Abbreviation: None.

Arguments: A list of symbols with optional value assignments separated by commas and enclosed within parentheses. For example:

SET (*symbol* [= *number*] [, *symbol* [= *number*] …])

Default: None.

Control Class: General

Description: The **SET** control assigns numeric values to the specified symbols. Symbols that are specified with an equal sign (=) and a numeric value are assigned the specified value. Symbols that do not include an explicit value assignment are assigned the value 0FFFFh.

These symbols can be used in **IF** and **ELSEIF** control statements for conditional assembly. They are only used for control of the assembler using the conditional assembly controls. These symbols are administered separately and do not interfere with the other code, bit, data and xdata symbols.

Example:

```
A251 SAMPLE.A51 SET(DEBUG1=1, DEBUG2=0, DEBUG3=1)

$SET (TESTCODE = 0)

$SET (DEBUGCODE, PRINTCODE)
```

RESET

Abbreviation: None.

Arguments: A list of symbols separated by commas and enclosed within parentheses. For example:

RESET (*symbol* [, *symbol* …])

Default: None

Control Class: General

Description: The **RESET** control assigns a value of 0000h to the specified symbols. These symbols can then be used in **IF** and **ELSEIF** control statements for conditional assembly. These symbols are only used for control of the assembler using the conditional assembly controls. They are administered separately and do not interfere with the other code, bit, data and xdata symbols.

Example:

```
A251 SAMPLE.A51 RESET(DEBUG1, DEBUG2, DEBUG3)

$RESET (TESTCODE)

$RESET (DEBUGCODE, PRINTCODE)
```

IF

Abbreviation: None

Arguments: A numeric expression

Default: None

Control Class: General

Description: The **IF** control begins an **IF**–**ELSE**–**ENDIF** construct that is used for conditional assembly of 8051 program code. The specified numeric expression is evaluated and if it is non–zero (TRUE) the **IF** block is assembled. If the expression is zero (FALSE), the **IF** block is not assembled and the subsequent blocks of the **IF** construct are evaluated.

IF blocks can be terminated by an **ELSE**, **ELSEIF**, or **ENDIF** control statement.

Example:

```
.
.
.
$IF (DEBUG_VAR = 3)
.
.
.
Version_3:      MOV   DPTR, #TABLE_3
.
.
.
$ ENDIF
.
.
.
```

ELSEIF

Abbreviation: None

Arguments: A numeric expression.

Default: None

Description: The **ELSEIF** control is used to introduce an alternative program block after an **IF** or **ELSEIF** control. The **ELSEIF** block is only assembled if the specified numeric expression is non–zero (TRUE) and if previous **IF** and **ELSEIF** conditions in the **IF**–**ELSE**–**ENDIF** construct were FALSE. **ELSEIF** blocks are terminated by an **ELSEIF**, **ELSE**, or **ENDIF** control.

Example:

```
.
.
.
$IF SWITCH = 1                ; Assemble if switch is 1
.
.
.
$ELSEIF SWITCH = 2            ; Assemble if switch is 2
.
.
.
$ELSEIF SWITCH = 3            ; Assemble if switch is 3
.
.
.
$ENDIF
.
.
.
```

ELSE

Abbreviation: None.

Arguments: None.

Default: None.

Control Class: General

Description: The **ELSE** control is used to introduce an alternative program block after an **IF** or **ELSEIF** control. The **ELSE** block is only assembled if previous **IF** and **ELSEIF** conditions in the **IF**–**ELSE**–**ENDIF** construct were all FALSE. **ELSE** blocks are terminated with an **ENDIF** control.

Example:

```
.
.
.
$IF (DEBUG)       ; TRUE when DEBUG <> 1
.
.
.
$ELSEIF (TEST)
.
.
.
$ELSE
.
.
.
$ENDIF
.
.
.
```

ENDIF

Abbreviation: None

Arguments: None

Default: None

Control Class: General

Description: The **ENDIF** control terminates an **IF**–**ELSE**–**ENDIF** construct. When A251 encounters an **ENDIF** control statement, it concludes processing the **IF** block and resumes assembly at the point of the **IF** block. Since **IF** blocks can be nested, this may involve continuing in another **IF** block. The **ENDIF** control must be preceded by an **IF**, **ELSEIF**, or **ELSE** control block.

Example:

```
.
.
.
$IF TEST
.
.
.
$ENDIF
.
.
.
```

Chapter 8. Error Messages

This chapter lists the error messages generated by A251. The following sections include a brief description of the possible error messages along with a description of the error and any corrective actions you can take to avoid or eliminate the error.

Fatal errors terminate the assembly and generate a message that is displayed on the console. Non–fatal errors generate a message in the assembly listing file but do not terminate the assembly.

Fatal Errors

Fatal errors cause immediate termination of the assembly. These errors usually occur as a result of an invalid command line. Fatal errors are also generated when the assembler cannot access a specified source file or when the macros are nested more than 9 deep.

Fatal errors produce a message that conforms to one of the following formats:

```
A251 FATAL ERROR –
    FILE:                    <file in which the error occurred>
    LINE:                    <line in which the error occurred
    ERROR:                   <corresponding error message>
A251 TERMINATED.
```

or

```
A251 FATAL ERROR –
    ERROR:                   <error message with description>
A251 TERMINATED.
```

where

FILE	is the name of an input file that could not be opened.
LINE	is the line where the error occurred
ERROR	is the fatal error message text explained below.

8

Fatal Error Messages

ATTEMPT TO SHARE FILE
A file is used both for input and output (e.g. list file uses the same name as the source file).

BAD NUMERIC CONSTANT
The numeric argument to the given control is illegal.

CAN'T ATTACH FILE
The given file can't be opened for read access.

CAN'T CREATE FILE
The given file can't be opened for write/update access.

CAN'T HAVE GENERAL CONTROL IN INVOCATION LINE
The given control is allowed in $control lines within the source file only (for example the **EJECT** control). Some controls are allowed only in the source text and not in the command line. Refer to "Chapter 7. Invocation and Controls" on page 113 for more information about the A251 controls.

CAN'T REMOVE FILE
The given temporary file could not be removed for some reason.

CONFLICTING CONTROL
The given control conflicts with an earlier control (for example **$NOMOD251 MODSRC**).

CONTROL LINE TOO LONG (500)
A $-control line has more than 500 characters.

DISK FILE REQUIRED
The given file does not represent a disk file.

ERRORPRINT– AND LIST–FILE CANNOT BE THE SAME
It is illegal to direct the listing file output and the errorprint output to the console at the same time.

EXPECTED DELIMITER '(' AFTER CONTROL
The given control requires a brace enclosed argument

EXPECTED DELIMITER ')' AFTER ARGUMENT
The given control requires a brace enclosed argument

FILE DOES NOT EXIST
The given file does not exist.

FILE IS READ ONLY
The given file does not permit write/update access.

FILE WRITE ERROR
The given file could not be written to (check free space)

IDENTIFIER EXPECTED
The given control requires an identifier as it's argument, for example **SET (VAR1=1234H).**

ILLEGAL FILE NAME, VOLUME OR DIRECTORY NAME
The name of the file is invalid or designates an invalid file.

INVOCATION LINE TOO LONG
The invocation line is longer than 500 characters.

LIMIT EXCEEDED: BALANCED TEXT LENGTH
The maximum length of a balanced text string is 65000 characters.

LIMIT EXCEEDED: INCLUDE OR MACRO NESTING
The maximum nesting level for MPL-macros is 50. The maximum nesting level of standard macros plus include files is 10.

LIMIT EXCEEDED: MACRO DEFINITION LENGTH
The maximum definition length of a standard macro is 20000 characters. MPL macros are limited to 65000 characters.

LIMIT EXCEEDED: MORE THAN 16000 SYMBOLS
The number of symbols (labels, equ/set symbols, externals, segment-symbols) must not exceed 16000 per source file.

LIMIT EXCEEDED: SOURCE LINE LENGTH (500)
A single source line must not exceed the 500 characters per line limit.

LIMIT EXCEEDED: TOO MANY EXTERNALS (65535)
The number of external symbols must not exceed 65535 per source module.

LIMIT EXCEEDED: TOO MANY EXTERNALS (65535)
The number of externals must not exceed 65535 per source module.

LIMIT EXCEEDED: TOO MANY SEGMENTS (65535)
The number of segments must not exceed 65535 per source module.

NON-NULL ARGUMENT EXPECTED
The argument to the given control must not be null (for example **$PRINT()**).

OUT OF MEMORY
The assembler has run out of memory. Remove unnecessary drivers from your system configuration.

OUT OF RANGE NUMERIC VALUE
The numeric argument to the given control is out of range (for example **$PAGEWIDTH(3000)**).

UNKNOWN CONTROL
The given control is undefined.

Non–Fatal Errors

Non–fatal errors usually occur within the source program and are typically syntax errors. When one of these errors is encountered, the assembler attempts to recover and continue processing the input file. As more errors are encountered, the assembler will produce additional error messages. The error messages that are generated are included in the listing file.

Non–fatal errors produce a message using the following format:

****** ERROR number IN line (file, LINE line): error message***

or

****** WARNING number IN line (file, LINE line): warning message***

where

number	is the error number.
line	corresponds to the logical line number in the source file.
file	corresponds to the source or include file which contains the error.
LINE	corresponds to the physical line number in <file>.
error message	is descriptive text and depends on the type of error encountered.

The logical line number in the source file is counted including the lines of all include files and may therefore differ from the physical line number. For that reason, the physical line number and the associated source or include file is also given in error and warning messages.

Example

```
       11               MOV     R0,# 25 * | 10
*** ---------------------------------------^
*** ERROR #4 IN 11 (TEST.A51, LINE 11), ILLEGAL CHARACTER
```

The caret character (^) is used to indicate the position of the incorrect character or to identify the point at which the error was detected. It is possible that the position indicated is due to a previous error. If a source line contains more than one error, the additional position indicators are displayed on subsequent lines.

The following table lists the non–fatal error messages that are generated by A251. These messages are listed by error number along with the error message and a brief description of possible causes and corrections.

NOTE
Errors marked by † are MCS 251 specific and are not generated by the A51 assembler.

Number	Non–Fatal Error Message and Description
1	**ILLEGAL CHARACTER IN NUMERIC CONSTANT** This error indicates that an invalid character was found in a numeric constant. Numeric constants must begin with a decimal digit and are delimited by the first non–numeric character (with the exception of the dollar sign). The base of the number decides which characters are valid. • Base 2: 0, 1 and the base indicator B • Base 8: 0–7 and the base indicator O or Q • Base 10: 0–9 and the base indicator D or no indicator • Base 16: 0–9, A–F and the base indicator H • Base 16: 0xhhhh, 0–9, and A–F
2	**MISSING STRING TERMINATOR** The ending string terminator was missing. The string was terminated with a carriage return.
3	**ILLEGAL CHARACTER** The assembler has detected a character which is not in the set of valid characters for the 51/251 assembler language (for example `).
4	**BAD INDIRECT REGISTER IDENTIFIER** This error occurs if combined registers are entered incorrectly; e.g., @R7, @R3, @PC+A, @DPTR+A.
5	**ILLEGAL USE OF A RESERVED WORD** This error indicates that a reserved word is used for a label.

Number	Non–Fatal Error Message and Description
6	**DEFINITION STATEMENT EXPECTED** The first symbol in the line must be part of a definition. For example: VAR1 EQU 12
7	**LABEL NOT PERMITTED** A label was detected in an invalid context.
8	**ATTEMPT TO DEFINE AN ALREADY DEFINED LABEL** A label was defined more than once. Labels may be defined only once in the source program.
9	**SYNTAX ERROR** A51/A251 encountered an error processing the line at the specified token.
10	**ATTEMPT TO DEFINE AN ALREADY DEFINED SYMBOL** An attempt was made to define a symbol more than once. The subsequent definition was ignored.
11	**STRING CONTAINS ZERO OR MORE THAN TWO CHARACTERS** Strings used in an expression can be a maximum of two characters long (16 bits).
12	**ILLEGAL OPERAND** An operand was expected but was not found in an arithmetic expression. The expression is illegal.
13	**')' EXPECTED** A right parenthesis is expected. This usually indicates an error in the definition of external symbols.
14	**BAD RELOCATABLE EXPRESSION** A relocatable expression may contain only one relocatable symbol which may be a segment symbol, external symbol, or a symbol belonging to a relocatable segment. Mathematical operations cannot be carried out on more than one relocatable symbol.
15	**MISSING FACTOR** A constant or a symbolic value is expected after an operator.
16	**DIVIDE BY ZERO ERROR** A division by zero was attempted while calculating an expression. The value calculated is undefined.

Number	Non–Fatal Error Message and Description
17	**INVALID BASE IN BIT ADDRESS EXPRESSION** This error indicates that the byte base in the bit address is invalid. This occurs if the base is outside of the range 20h–2Fh or if it lies between 80h and 0FFh and is not evenly divisible by 8. For the 251 chip, the byte base address must be in range 20H-0FFH with no restrictions. Note that with symbolic operands, the operand specifies an absolute bit segment or an addressable data segment.
18	**OUT OF RANGE OR NON–TYPELESS BIT–OFFSET** The input of the offset (base.offset) in a bit address must be a typeless absolute expression with a value between 0 and 7.
19	**INVALID REGISTER FOR EQU/SET** The registers R0–R7, A and C may be used in SET or EQU directives. No other registers are allowed.
20	**INVALID SIMPLE RELOCATABLE EXPRESSION** A simple relocatable expression is intended to represent an address in a relocatable segment. External symbols as well as segment symbols are not allowed. The expression however may contain more symbols of the same segment. Simple relocatable expressions are allowed in the instructions ORG, EQU, SET, CODE, XDATA, IDATA, BIT, DATA, DB and DW.
21	**EXPRESSION WITH FORWARD REFERENCE NOT PERMITTED** Expressions in EQU and SET directives may not contain forward references.
22	**EXPRESSION TYPE DOES NOT MATCH INSTRUCTION** The expression does not conform to the 8051/251 conventions. A #, /Bit, register, or numeric expression was expected.
23	**NUMERIC EXPRESSION EXPECTED** A numeric expression is expected. The expression of another type is found.
24	**SEGMENT–TYPE EXPECTED** The segment type of a definition was missing or invalid.
25	**RELOCATION–TYPE EXPECTED** An invalid relocation type for a segment definition was encountered. One of the following is valid: UNIT, PAGE, INPAGE, INBLOCK and BITADDRESSABLE (for A251: same as before plus INSEG and EBITADDRESSABLE).

Number	Non–Fatal Error Message and Description
26	**INVALID RELOCATION–TYPE** The types PAGE and INPAGE are only allowed for the CODE/ECODE and XDATA segments. INBLOCK/INSEG is only allowed for the CODE/ECODE segments and BITADDRESSABLE is only for the DATA segment (maximum length 16 Bytes). EBITADDRESSABLE is allowed for DATA segments (maximum length 96 Bytes). The type UNIT is the default for all segment types if no input is entered.
27	**LOCATION COUNTER MAY NOT POINT BELOW SEGMENT–BASE** An ORG directive used in a segment defined by the AT address directive may not specify an offset that lies below the segment base. The following example is, therefore, invalid: CSEG AT 1000H ORG 800H
28	**ABSOLUTE EXPRESSION REQUIRED** The expression in a DS or DBIT instruction must be an absolute typeless expression. Relocatable expressions are not allowed.
29	**SEGMENT–LIMIT EXCEEDED** The maximum limit of a segment was exceeded. This limit depends on the segment and relocation type. Segments with the attribute DATA should not exceed 128 bytes. BITADDRESSABLE segments should not exceed 16 bytes and INPAGE segments should not exceed 2 KBytes.
30	**SEGMENT–SYMBOL EXPECTED** The operand to an RSEG directive must be a segment symbol.
31	**PUBLIC–ATTRIBUTE NOT PERMITTED** The PUBLIC attribute is not allowed on the specified symbol.
32	**ATTEMPT TO RESPECIFY MODULE NAME** An attempt was made to redefine the name of the module by using a second NAME directive. The NAME directive may only appear once in a program.
33	**CONFLICTING ATTRIBUTES** A symbol may not contain the attributes PUBLIC and EXTRN simultaneously.
34	**',' EXPECTED** A comma is expected in a list of expressions or symbols.
35	**'(' EXPECTED** A left parenthesis is expected at the indicated position.

Number	Non–Fatal Error Message and Description
36	**INVALID NUMBER FOR REGISTERBANK** The expression in a REGISTERBANK control must be an absolute typeless number between 0 and 3.
37	**OPERATION INVALID IN THIS SEGMENT** 8051/251 instructions are allowed only within CODE/ECODE segments.
38	**NUMBER OF OPERANDS DOES NOT MATCH INSTRUCTION** Either too few or too many operands were specified for the indicated instruction. The instruction was ignored.
39	**REGISTER–OPERAND EXPECTED** A register operand was expected but an operand of another type was found.
40	**INVALID REGISTER** The specified register operand does not conform to the 8051/251 conventions.
41	**MISSING 'END' STATEMENT** The last instruction in a source program must be the END directive. The preceding source is assembled correctly and the object is valid.
42	**INTERNAL ERROR (PASS-2), CONTACT TECHNICAL SUPPORT** This error occurs if a symbol in Pass 2 contains a value different than in Pass 1.
43	**RESPECIFIED PRIMARY CONTROL, LINE IGNORED** A control was repeated or conflicts with a previous control. The control statement was ignored.
44	**MISPLACED PRIMARY CONTROL, LINE IGNORED** A primary control was misplaced. Primary controls may be entered in the invocation line or at the beginning of the source file (as $ instruction). The processing of primary controls in a source file ends when the first non empty/non comment line containing anything but a primary control is processed.
45	**UNDEFINED SYMBOL (PASS–2)** The symbol is undefined.
46	**CODE/ECODE–ADDRESS EXPECTED** An operand of memory type CODE/ECODE or a typeless expression is expected.

Number	Non–Fatal Error Message and Description
47	**XDATA–ADDRESS EXPECTED** An operand of memory type XDATA or a typeless expression is expected.
48	**DATA–ADDRESS EXPECTED** An operand of memory type DATA or a typeless expression is expected.
49	**IDATA–ADDRESS EXPECTED** An operand of memory type 'IDATA' or a typeless expression is expected.
50	**BIT–ADDRESS EXPECTED** An operand of memory type BIT or a typeless expression is expected.
51	**TARGET OUT OF RANGE** The target of a conditional jump instruction is outside of the +127/–128 range or the target of an AJMP or ACALL instruction is outside the 2 KByte memory block.
52	**VALUE HAS BEEN TRUNCATED TO 8 BITS** The result of the expression exceeds 255 decimal. Only the 8 low–order bits are used for the byte operand.
53	**MISSING 'USING' INFORMATION** The absolute register symbols AR0 through AR7 can be used only if a USING registerbank directive was specified. This error indicates that the USING directive is missing and the assembler cannot assign data addresses to the register symbols.
54	**MISPLACED CONDITIONAL CONTROL** An ELSEIF, ELSE, or ENDIF control must be preceded by an IF instruction.
55	**BAD CONDITIONAL EXPRESSION** The expression to the IF or ELSEIF control is invalid. These expressions must be absolute and may not contain relocatable symbols. The $IF and $ELSEIF can only access symbols defined with the $SET and $RESET controls. Both IF and ELSEIF allow access to all symbols of the source program.
56	**UNBALANCED IF–ENDIF–CONTROLS** Each IF block must be terminated with an ENDIF control. This is also true with skipped nested IF blocks.
57	**SAVE STACK UNDERFLOW** A $RESTORE control instruction is then valid only if a $SAVE control was previously given.

Number	Non–Fatal Error Message and Description
58	**SAVE STACK OVERFLOW** The context of the GEN, **COND**, and LIST controls may be stored by the $SAVE control up to a maximum of 9 levels.
59	**MACRO REDEFINITION** An attempt was made to define an already defined macro.
60	**ERROR-60** Not generated by A51/251.
61	**MACRO TERMINATED BY END OF FILE, MISSING 'ENDM'** An attempt was made to define an already defined macro.
62	**TOO MANY FORMAL PARAMETERS (16)** The number of formal parameters to a macro is limited to 16.
63	**TOO MANY LOCALS (16)** The number of local symbols within a macro is limited to 16.
64	**DUPLICATE LOCAL/FORMAL** The number of local or formal identifier must be distinct.
65	**IDENTIFIER EXPECTED** While parsing a macro definition, an identifier was expected but something different was found.
66	**'EXITM' INVALID OUTSIDE A MACRO** The EXITM (exit macro) keyword is illegal outside a macro definition.
67	**EXPRESSION TOO COMPLEX** A too complex expression was encountered. This error occurs, if the number of operands and operators in one expression exceeds 50.
68	**UNKNOWN CONTROL OR BAD ARGUMENT(S)** The control given in a $-control line or the argument(s) to some control are invalid.
69	**MISPLACED ELSEIF/ELSE/ENDIF CONTROL** These controls require a preceding IF control.
70	**LIMIT EXCEEDED: IF-NESTING (10)** IF controls may be nested up to a level of 10.

Number	Non–Fatal Error Message and Description
71	**NUMERIC VALUE OUT OF RANGE** The value of a numeric expression is out of range (for example $PAGEWIDTH (2048) where only values in range 80 to 132 are allowed).
72	**TOO MANY TOKENS IN SOURCE LINE** The number of tokens (identifiers, operators, punctuation characters and end of line) exceeds 200. The source line is truncated at 200 tokens.
72	**TOO MANY TOKENS IN SOURCE LINE** The number of tokens (identifiers, operators, punctuation characters and end of line) exceeds 200. The source line is truncated at 200 tokens.
73	**TEXT FOUND BEYOND END STATEMENT - IGNORED** Text following the END directive is not processed by the assembler.
74	**REGISTER USAGE: UNDEFINED REGISTER NAME** A register name argument given to the REGUSE control does not represent the name of a register.
75	**'REGISTER USAGE' REQUIRES A PUBLIC CODE SYMBOL** The register usage value must be assigned to a public symbol, which represents a CODE symbol. For the A251, the name of a public procedure (near of far) with memory type CODE/ECODE is also valid.
76	**MULTIPLE REGISTER USES GIVEN TO ONE SYMBOL** The register usage value may be assigned to a symbol or procedure only once.
77	**INSTRUCTION NOT AVAILABLE** † The given instruction is not available in the current mode of operation.
78	**ERROR 78** Not generated by A51/251.
79	**INVALID ATTRIBUTE** † The OVERLAYABLE attribute given in a segment definition is not valid for code and constant segments.
80	**INVALID ABSOLUTE BASE/OFFS VALUE** † The absolute address given in a segment definition does not conform to the memory type of the segment (for example DATA AT 0x1000).

† New features in the A251 assembler and the MCS 251 architecture

Number	Non–Fatal Error Message and Description
81	**EXPRESSION HAS DIFFERENT MEMORY SPACE** † The expression given in a symbol definition statement does not have the memory space required by the directive, for example: VAR1 CODE EXPR where 'EXPR' has a memory type other than CODE or NUMBER.
82	**LABEL STATEMENT MUST BE WITHIN CODE/ECODE SEGMENT** † The LABEL statement is not allowed outside a CODE or ECODE segment.
83	**TYPE INCOMPATIBLE WITH GIVEN MEMORY SPACE** † The type given in an external declaration is not compatible to the given memory space. The following examples shows an invalid type since a bit can never reside in code space: EXTRN CODE:BIT (bit0, bit1)
84	**OPERATOR REQUIRES A CODE/ECODE ADDRESS** † The type override operators NEAR and FAR cannot be applied to addresses with memory type other than CODE and ECODE.
85	**INVALID OPERAND TYPE** † An expression contains invalid typed operands to some operator, for example addition/unary minus on bit-type operands.
86	**PROCEDURES CAN'T BE NESTED** † A251 does not support nested procedures.
87	**UNCLOSED PROCEDURE** † A251 detected an unclosed procedure after scanning the source file.
88	**VALUE HAS BEEN TRUNCATED TO 16 BITS** † The displacement value given in a register expression (WRn+disp16, DRk+disp16) has been truncated to 16 bits.
89	**ERROR-89** † Not generated by A51/251.
90	**'FAR' RETURN IN 'NEAR' PROCEDURE** † The return far instruction (ERET) was encountered in a procedure of type NEAR (the code may not work).

Number	Non–Fatal Error Message and Description
91	**TYPE MISMATCH** † The operand type of an instruction operand does not match the requested type of the instruction, for example: MOV WR10,Byte_Memory_Operand. ; Word/Byte mismatch Use a type override to avoid the warning as shown: MOV WR10,WORD Byte_Memory_Operand
92	**MCS 251 INSTRUCTION IN NON 251 MODE** † The assembler encountered an MCS 251 instruction in $NOMOD251 mode of operation. $NOMOD251 limits the instructions to the set for the MCS 51 family of controllers.
93	**ERROR-93** Not generated by A51/251.
94	**VALUE DOES NOT MATCH INSTRUCTION** † The short value given in a INC/DEC Rn,#short is not one of 1,2,4.
95	**ILLEGAL MEMORY CLASS SPECIFIER** † The memory class specifier in a segment definition statement does not correspond to one of the predefined memory class names (CODE, ECODE, BIT, EBIT ...).
96	**ACCESS TO MISALIGNED ADDRESS** † A word instruction accesses a misaligned (odd) address. This warning is generated only if the $WORDALIGN control was given.
97	**'FAR' REFERENCE TO 'NEAR' LABEL** † An ECALL/AJMP instruction to some label of type NEAR has been detected.
98	**'NEAR' REFERENCE TO 'FAR' LABEL** † An ACALL/AJMP/SJMP or conditional jump instruction to some label of type FAR has been detected.
150	**PREMATURE END OF FILE ENCOUNTERED** The MPL macro processor encountered the end of the source file while parsing a macro definition.
151	**<name>: IDENTIFIER EXPECTED** The macro or function given by <name> in the error message expected an identifier but found something else.

† New features in the A251 assembler and the MCS 251 architecture

Number	Non–Fatal Error Message and Description
152	**MPL FUNCTION <name>: <character> EXPECTED** The MPL function <name> expected a specific character in the input stream but found some other character.
153	**<name>: UNBALANCED PARENTHESIS** While scanning balanced text, the macro processor expected a ‘)’ character, but found some other character.
154	**EXPECTED <identifier>** The macro processor expected some specific identifier (for example ELSE) but found some other text.
155	**ERROR-155** Not generated by A51/251.
156	**FUNCTION ‘MATCH’: ILLEGAL CALL PATTERN** The call pattern to the MPL function match must be a formal parameter followed by a delimiter followed by another formal parameter.
157	**FUNCTION ‘EXIT’ IN BAD CONTEXT** The EXIT function must not appear outside a macro expansion, %REPEAT or %WHILE.
158	**ILLEGAL METACHARACTER <character>** The metacharacter may not be @, (,), *, TAB, EOL, A-Z,a-z, 0-9, _ and ?.
159	**CALL PATTERN - DELIMITER <delimiter> NOT FOUND** The actual parameters in a macro call do not match the call pattern defined in the macro definition of that macro.
160	**CALL TO UNDEFINED MACRO <macroname>** An attempt to activate an undefined macro has been encountered .
161	**ERROR-161** Not generated by A51/251.
162	**INVALID DIGIT ‘character’ IN NUMBER** An ill formed number has been encountered. For numbers, the rules are equal to the numbers in the assembler language with the exception of $ signs, which are not supported within the MPL.
163	**UNCLOSED STRING OR CHARACTER CONSTANT** A string or character constant is terminated by an end of line character instead of the closing character.

Number	Non–Fatal Error Message and Description
164	**INVALID STRING OR CHARACTER CONSTANT** A string or character constant may contain one or two characters.
165	**EVAL: UNKNOWN EXPRESSION IDENTIFIER** An MPL expression contains an unknown identifier.
166	**<token>: INVALID EXPRESSION TOKEN** An MPL expression contains a token which neither represents an operator nor an operand.
167	**<function>: DIV/MOD BY ZERO** The evaluation of an expression within the MPL function <function> yields a division or modulus by zero.
168	**EVAL: SYNTAX ERROR IN EXPRESSION** An expression is followed by one or more erroneous tokens.
169	**CAN'T OPEN FILE <name of file>** The file given in an $INCLUDE directive cannot be opened.
170	**<name of file>: IS NOT A DISK FILE** An attempt was made to open a file which is not a disk file (for example $INCLUDE (CON).
171	**ERROR IN INCLUDE DIRECTIVE** The argument to the INCLUDE directive must be the brace enclosed name of the file, for example $INCLUDE (REG251.INC).
172	**CAN'T REDEFINE PREDEFINED MACRO 'SET'** The .predefined %SET macro can't be redefined.

Appendix A. 8051/251 Instruction Sets

A

This appendix lists the 8051 and MCS 251 microcontroller instruction sets. The 8051 and MCS 251 instructions are listed in alphabetical order and according to their hexadecimal opcodes. The following terms are used in the descriptions.

Identifier	Explanation
A	Accumulator
AB	Register Pair A & B
B	Multiplication Register
C	Carry Flag
DPTR	Data pointer
PC	Program Counter
Rn	Register R0 - R7 of the currently selected Register Bank.
Rm †	Register R0 - R15 of the currently selected Register File.
WRj †	Register WR0 - WR30 of the currently selected Register File.
DRk †	Register DR0 - DR28, DR56, DR60 of the currently selected Register File.
dir8	8-bit data address; Data RAM location (00:00 - 00:7F) or a SFR (S:80 - S:FF)
dir16 †	16-bit data address; Data RAM location (00:00 - 00:FFFF).
@Ri	Data RAM location (00:00 - 00:FF) addressed indirectly through R1 or R0.
@WRj †	Data RAM location (0 - 64K) addressed indirectly through WR0 - WR30.
@DRk †	Data RAM location (0 - 16M) addressed indirectly through register DR60, DR56, DR0 - DR28.
#data8	8-bit constant included in instruction.
#data16	16-bit constant included in instruction.
#short †	constant 1, 2 or 4 included in instruction (251 only).
addr16	16-bit destination address. Used by LCALL & LJMP. A branch can be anywhere within a 64KB segment of the program memory address space.
addr11	11-bit destination address. Used by ACALL & AJMP. The branch will be within the same 2KByte block of program memory of the first byte of the following instruction.
rel	Signed (two's complement) 8-bit offset byte. Used by SJMP and conditional jumps. Range is -128 .. +127 bytes relative to the first byte of the following instruction.
bit8	Direct addressed bit in Data RAM Location (8051 compatible).
bit11 †	Direct addressed bit in Data RAM or Special Function Register.
@Wrj+dis †	Data RAM location (0 - 64K) addressed displaced through (WR0 - WR30) + displacement value (251 only).
@DRk+dis †	Data RAM location (0 - 16M) addressed displaced through (DR60, DR56, DR28 - DR0) + displacement value (251 only).

† New features in the A251 assembler and the MCS 251 architecture

A

ACALL	Absolute Subroutine CALL	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
ACALL	addr11	Absolute Subroutine Call	2	2

ADD	ADD destination, source Addition	CY	AC	N	OV	Z
		X	X	X	X	X

Mnemonic		Description	Bytes Binary	Bytes Source †
ADD	A,Rn	Add register to accumulator	1	2
ADD	A,dir8	Add direct byte to accumulator	2	2
ADD	A,@Ri	Add indirect RAM to accumulator	1	2
ADD	A,#data8	Add immediate data to accumulator	2	2
ADD	Rm,Rm †	Add byte register to byte register	3	2
ADD	WRj,WRj †	Add word register to word register	3	2
ADD	DRk,DRk †	Add double word register to dword register	3	2
ADD	Rm,#data8 †	Add 8 bit data to byte register	4	3
ADD	Wrj,#data16 †	Add 16 bit data to word register	5	4
ADD	Drk,#data16 †	Add 16 bit unsigned data to dword register	5	4
ADD	Rm,dir8 †	Add direct address to byte register	4	3
ADD	WRj,dir8 †	Add direct address to word register	4	3
ADD	Rm,dir16 †	Add direct address (64K) to byte register	5	4
ADD	WRj,dir16 †	Add direct address (64K) to word register	5	4
ADD	Rm,@WRj †	Add indirect address (64K) to byte register	4	3
ADD	Rm,@DRk †	Add indirect address (16M) to byte register	4	3

ADDC	ADDC destination, source Addition with Carry	CY	AC	N	OV	Z
		X	X	X	X	X

Mnemonic		Description	Bytes Binary	Bytes Source †
ADDC	A,Rn	Add register to accumulator with carry flag	1	2
ADDC	A,dir8	Add direct byte to accumulator with carry flag	2	2
ADDC	A,@Ri	Add indirect RAM to accumulator with carry flag	1	2
ADDC	A,#data8	Add immediate data to accumulator with carry flag	2	2

† New features in the A251 assembler and the MCS 251 architecture

A

AJMP	Absolute JUMP	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
AJMP	addr11	Absolute Jump	2	2

ANL	ANL destination, source Logical AND	CY	AC	N	OV	Z
		—	—	X	—	X

Mnemonic		Description	Bytes Binary	Bytes Source †
ANL	A,Rn	AND register to accumulator	1	2
ANL	A,dir8	AND direct byte to accumulator	2	2
ANL	A,@Ri	AND indirect RAM to accumulator	1	2
ANL	A,#data8	AND immediate data to accumulator	2	2
ANL	dir,A	AND accumulator to direct byte	2	2
ANL	dir,#data8	AND immediate data to direct byte	3	3
ANL	Rm,Rm †	AND byte register to byte register	3	2
ANL	WRj,WRj †	AND word register to word register	3	2
ANL	Rm,#data8 †	AND 8 bit data to byte register	4	3
ANL	Wrj,#data16 †	AND 16 bit data to word register	5	4
ANL	Rm,dir8 †	AND direct address to byte register	4	3
ANL	Wrj,dir8 †	AND direct address to word register	4	3
ANL	Rm,dir16 †	AND direct address (64K) to byte register	5	4
ANL	Wrj,dir16 †	AND direct address (64K) to word register	5	4
ANL	Rm,@WRj †	AND indirect address (64K) to byte register	4	3
ANL	Rm,@DRk †	AND indirect address (16M) to byte register	4	3

ANL	ANL destination, source Logical AND for bit variables	CY	AC	N	OV	Z
		X	—	X	—	X

Mnemonic		Description	Bytes Binary	Bytes Source †
ANL	C,bit8	AND direct bit to carry; from BIT space	2	2
ANL	C,bit11 †	AND direct bit to carry; from EBIT space	4	3

A

ANL/	ANL/ destination, source Logical AND for bit variables	CY X	AC —	N X	OV —	Z X

Mnemonic		Description	Bytes Binary	Bytes Source †
ANL	C,/bit8	AND complement of direct bit to carry; BIT space	2	2
ANL	C,/bit11 †	AND complement of dir bit to carry; EBIT space	4	3

CJNE	COMPARE destination, source and jump if not equal	CY X	AC —	N X	OV —	Z X

Mnemonic		Description	Bytes Binary	Bytes Source †
CJNE	A,dir8,rel	Compare dir byte to acc. and jump if not equal	3	3
CJNE	A,#data8,rel	Compare imm. data to acc. and jump if not equal	3	3
CJNE	Rn,#data8,rel	Compare imm. data to reg and jump if not equal	3	4
CJNE	@Ri,#data8,rel	Compare imm. data to indir and jump if not equal	3	4

CLR	CLEAR Operand	CY —	AC —	N X	OV —	Z X

Mnemonic		Description	Bytes Binary	Bytes Source †
CLR	bit11 †	Clear accumulator	1	1

CLR	CLEAR Bit Operand	CY —	AC —	N —	OV —	Z —

Mnemonic		Description	Bytes Binary	Bytes Source †
CLR	C	Clear carry	1	1
CLR	bit8	Clear direct bit from BIT space	2	2
CLR	bit11 †	Clear direct bit from EBIT space	4	3

† New features in the A251 assembler and the MCS 251 architecture

A

CMP	COMPARE Operands	CY X	AC X	N X	OV X	Z X

Mnemonic		Description	Bytes Binary	Bytes Source †
CMP	Rm,Rm †	Compare registers	3	2
CMP	WRj,WRj †	Compare word registers	3	2
CMP	DRk,DRk †	Compare double word registers	3	2
CMP	Rm,#data8 †	Compare register with immediate data	4	3
CMP	Wrj,#data16 †	Compare word register with immediate data	5	4
CMP	Drk,#00&#16 †	Compare dword reg with zero extended data	5	4
CMP	Drk,#ff&#16 †	Compare dword reg with one extended data	5	4
CMP	Rm,dir8 †	Compare register with direct byte	4	3
CMP	WRj,dir8 †	Compare word register with direct word	4	3
CMP	Rm,dir16 †	Compar register with direct byte (64K)	5	4
CMP	WRj,dir16 †	Compare word register with direct byte (64K)	5	4
CMP	Rm,@WRj †	Compare register with indirect address (64K)	4	3
CMP	Rm,@DRk †	Compare register with indirect address (16M)	4	3

CPL	COMPLEMENT Operand	CY —	AC —	N X	OV —	Z X

Mnemonic		Description	Bytes Binary	Bytes Source †
CPL	A	Complement accumulator	1	1

CPL	COMPLEMENT Bit Operand	CY —	AC —	N —	OV —	Z —

Mnemonic		Description	Bytes Binary	Bytes Source †
CPL	C	Complement carry	1	1
CPL	bit8	Complement direct bit from BIT space	2	2
CPL	bit11 †	Complement direct bit from EBIT space	4	3

A

DA	DECIMAL ADJUST Accumulator for Addition	CY	AC	N	OV	Z
		X	—	X	—	X

Mnemonic		Description	Bytes Binary	Bytes Source †
DA	A	Decimal adjust accumulator	1	1

DEC	DECREMENT Operand with a constant	CY	AC	N	OV	Z
		—	—	X	—	X

Mnemonic		Description	Bytes Binary	Bytes Source †
DEC	A	Decrement accumulator	1	1
DEC	Rn	Decrement register	1	2
DEC	dir	Decrement dir byte	2	2
DEC	@Ri	Decrement indir RAM	1	2
DEC	Rm,#short †	Decrement byte register with 1, 2 or 4	3	2
DEC	WRj,#short †	Decrement word register with 1, 2 or 4	3	2
DEC	DRk,#short †	Decrement double word register with 1, 2 or 4	3	2

DIV	DIVIDE Operands	CY	AC	N	OV	Z
		0	—	X	X	X

Mnemonic		Description	Bytes Binary	Bytes Source †
DIV	AB	Divide A by B	1	1
DIV	Rm,Rm †	Divide byte register by byte register	3	2
DIV	WRj,WRj †	Divide word register by word register	3	2

DJNZ	DECREMENT Operand and Jump if Not Zero	CY	AC	N	OV	Z
		—	—	X	—	X

Mnemonic		Description	Bytes Binary	Bytes Source †
DJNZ	Rn,rel	Decrement register and jump if not zero	3	3
DJNZ	dir8,rel	Decrement direct byte and jump if not zero	3	3

A

ECALL	Extended Subroutine CALL	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
ECALL	addr24	Extended subroutine call	5	4
ECALL	DRk	Extended subroutine call	3	2

EJMP	Extended JUMP	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
EJMP	addr24	Extended jump	5	4
EJMP	DRk	Extended jump	3	2

ERET	RETURN from extended Subroutine	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
ERET	DRk	Return from subroutine	1	1

INC	INCREMENT Operand with a constant	CY	AC	N	OV	Z
		—	—	X	—	X

Mnemonic		Description	Bytes Binary	Bytes Source †
INC	A	Increment accumulator	1	1
INC	Rn	Increment register	1	2
INC	dir	Increment direct byte	2	2
INC	@Ri	Increment indirect RAM	1	2
INC	Rm,#short †	Increment byte register with 1, 2 or 4	3	2
INC	WRj,#short †	Increment word register with 1, 2 or 4	3	2
INC	Drk,#short †	Increment double word register with 1, 2 or 4	3	2
INC	DPTR	Increment Data Pointer	1	1

A

JB	JUMP if Bit is set	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
JB	bit8,rel	Jump if dir bit (from BIT space) is set	3	3
JB	bit11,rel †	Jump if dir bit (from EBIT space) is set	5	4

JBC	JUMP if Bit is set and clear bit	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
JBC	bit8,rel	Jump if dir bit (BIT space) is set and clear bit	3	3
JBC	bit11,rel †	Jump if dir bit (EBIT space) is set and clear bit	5	4

JC	JUMP if Carry is set	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
JC	rel	Jump if carry is set	2	2

JE	JUMP if equal	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
JE	rel †	Jump if equal	3	2

JG	JUMP if greater than	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
JG	rel †	Jump if greater than	3	2

A

JLE	JUMP if less than or equal	CY —	AC —	N —	OV —	Z —

Mnemonic		Description	Bytes Binary	Bytes Source †
JLE	rel †	Jump if less than or qual	3	2

JMP	JUMP indir relative to DPTR	CY —	AC —	N —	OV —	Z —

Mnemonic		Description	Bytes Binary	Bytes Source †
JMP	@A+DPTR	Jump indir relative to DPTR	1	1

JNB	JUMP if Bit is Not set	CY —	AC —	N —	OV —	Z —

Mnemonic		Description	Bytes Binary	Bytes Source †
JNB	bit8,rel	Jump if dir bit (from BIT space) is not set	3	3
JNB	bit11,rel †	Jump if dir bit (from EBIT space) is not set	5	4

JNC	JUMP if Carry is Not set	CY —	AC —	N —	OV —	Z —

Mnemonic		Description	Bytes Binary	Bytes Source †
JNC	rel	Jump if carry is not set	2	2

JNE	JUMP if Not Equal	CY —	AC —	N —	OV —	Z —

Mnemonic		Description	Bytes Binary	Bytes Source †
JNE	rel †	Jump if not equal	3	2

A

JNZ	JUMP if Accumulator is Not Zero	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
JNZ	rel	Jump if accumulator is not zero	2	2

JSG	JUMP if greater than (Signed)	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
JSG	rel †	Jump if greater than (signed)	3	2

JSGE	JUMP if greater than or Equal (Signed)	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
JSGE	rel †	Jump if greater than or equal (signed)	3	2

JSL	JUMP if Less than (Signed)	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
JSL	rel †	Jump if less than (signed)	3	2

JSLE	JUMP if Less than or Equal (Signed)	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
JSLE	rel †	Jump if less than or equal (signed)	3	2

A

JZ		JUMP if Accumulator is Zero	CY —	AC —	N —	OV —	Z —
Mnemonic		**Description**				**Bytes Binary**	**Bytes Source †**
JZ	rel	Jump if accumulator is zero				2	2

LCALL		Long Subroutine CALL	CY —	AC —	N —	OV —	Z —
Mnemonic		**Description**				**Bytes Binary**	**Bytes Source †**
LCALL	@WRj †	Long Subroutine Call indirect via word register				3	2
LCALL	addr16	Long Subroutine Call				3	3

LJMP		Long JUMP	CY —	AC —	N —	OV —	Z —
Mnemonic		**Description**				**Bytes Binary**	**Bytes Source †**
LJMP	@WRj †	Long Jump indirect via word register				3	2
LJMP	addr16	Long Jump				3	3

MOV		MOV destination, source Move data	CY —	AC —	N —	OV —	Z —
Mnemonic		**Description**				**Bytes Binary**	**Bytes Source †**
MOV	A,Rn	Move register to accumulator				1	2
MOV	A,dir8	Move direct byte to accumulator				2	2
MOV	A,@Ri	Move indirect RAM to accumulator				1	2
MOV	A,#data8	Move immediate data to accumulator				2	2
MOV	Rn,A	Move accumulator to register				1	2
MOV	Rn,dir8	Move direct byte to register				2	3
MOV	Rn,#data8	Move immediate data to register				2	3
MOV	dir8,A	Move accumulator to direct byte				2	2
MOV	dir8,Rn	Move register to direct byte				2	3
MOV	dir8,dir8	Move direct byte to direct byte				3	3
MOV	dir8,@Ri	Move indirect RAM to direct byte				2	3

† New features in the A251 assembler and the MCS 251 architecture

A

MOV	MOV destination, source Move data	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
MOV	dir8,#data8	Move immediate data to direct byte	3	3
MOV	@Ri,A	Move accumulator to indirect RAM	1	2
MOV	@Ri,dir8	Move direct byte to indirect RAM	2	3
MOV	@Ri,#data8	Move immediate data to indirect RAM	2	3
MOV	DPTR,#data16	Load Data Pointer with 16-bit constant	3	3
MOV	Rm,Rm †	Move byte register to byte register	3	2
MOV	WRj,WRj †	Move word register to word register	3	2
MOV	DRk,DRk †	Move dword register to dword register	3	2
MOV	Rm,#data8 †	Move 8 bit data to byte register	4	3
MOV	WRj,#data16 †	Move 16 bit data to word register	5	4
MOV	DRk,#0data16 †	Move 16 bit zero extended data to dword reg.	5	4
MOV	DRk,#1data16 †	Move 16 bit one extended data to dword reg.	5	4
MOV	Rm,dir8 †	Move dir address to byte register	4	3
MOV	WRj,dir8 †	Move direct address to word register	4	3
MOV	DRk,dir8 †	Move direct address to dword register	4	3
MOV	Rm,dir16 †	Move direct address (64K) to byte register	5	4
MOV	WRj,dir16 †	Move direct address (64K) to word register	5	4
MOV	DRk,dir16 †	Move direct address (64K) to dword register	5	4
MOV	Rm,@WRj †	Move indirect address (64K) to byte register	4	3
MOV	Rm,@DRk †	Move indirect address (16M) to byte register	4	3
MOV	WRj,@WRj †	Move indirect address (64K) to word register	4	3
MOV	WRj,@DRk †	Move indirect address (16M) to word register	4	3
MOV	dir8,Rm †	Move byte register to direct address	4	3
MOV	dir8,WRj †	Move word register to direct address	4	3
MOV	dir8,DRk †	Move dword register to direct address	4	3
MOV	dir16,Rm †	Move byte register to direct address (64K)	5	4
MOV	dir16,WRj †	Move word register to direct address (64K)	5	4
MOV	dir16,DRk †	Move dword register to direct address (64K)	5	4
MOV	@WRj,Rm †	Move byte register to direct address (64K)	4	3
MOV	@DRk,Rm †	Move byte register to indirect address (16M)	4	3
MOV	@WRj,WRj †	Move word register to indirect address (64K)	4	3
MOV	@DRk,WRj †	Move word register to indirect address (16M)	4	3
MOV	Rm,@WRj+dis †	Move displacement address (64K) to byte reg.	5	4
MOV	WRj,@WRj+dis †	Move displacement address (64K) to word reg.	5	4
MOV	Rm,@DRk+dis †	Move displacement address (16M) to byte reg.	5	4

† New features in the A251 assembler and the MCS 251 architecture

MOV	MOV destination, source Move data	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
MOV	WRj,@DRk+dis †	Move displacement address (16M) to word reg.	5	4
MOV	@WRj+dis,Rm †	Move byte reg. to displacement address (64K)	5	4
MOV	@WRj+dis,WRj †	Move word reg. to displacement address (64K)	5	4
MOV	@DRk+dis,Rm †	Move byte reg. to displacement address (16M)	5	4
MOV	@DRk+dis,WRj †	Move word reg. to displacement address (16M)	5	4
MOV	C,bit8	Move dir bit to carry	2	2
MOV	C,bit11 †	Move dir bit from 8 bit address location to carry	2	2
MOV	bit8,C	Move carry to dir bit	2	2
MOV	bit11,C †	Move carry to dir bit from 16 bit address location	5	4

MOVC	MOV destination, source Move Code byte	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
MOVC	A,@A+DPTR	Move code byte relative to DPTR to accumulator	1	1
MOVC	A,@A+PC	Move code byte relative to PC to accumulator	1	1

MOVH	MOVH destination, source Move data to high word of DR	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
MOVH	DRk,#data16 †	Move 16-bit imm. data to high word of dword reg.	3	2

A

MOVS	MOVS destination, source Move byte to word (signed ext.)	CY —	AC —	N —	OV —	Z —

Mnemonic		Description	Bytes Binary	Bytes Source †
MOVS	WRj,Rm †	Move byte register to word register	3	2

MOVX	MOV destination, source External RAM access	CY —	AC —	N —	OV —	Z —

Mnemonic		Description	Bytes Binary	Bytes Source †
MOVX	A,@Ri	Move xdata RAM (8 bit address) to accumulator	1	2
MOVX	A,@DPTR	Move xdata RAM (16 bit address) to accumulator	1	1
MOVX	@Ri,A	Move accumulator to xdata RAM (8 bit address)	1	2
MOVX	@DPTR,A	Move accumulator to xdata RAM (16 bit address)	1	1

MOVZ	MOV destination, source Move byte to word (zero ext.)	CY —	AC —	N —	OV —	Z —

Mnemonic		Description	Bytes Binary	Bytes Source †
MOVZ	WRj,Rm †	Move byte reg. to word reg. (zero extended)	3	2

MUL	MULTIPLY Operands	CY 0	AC —	N X	OV X	Z X

Mnemonic		Description	Bytes Binary	Bytes Source †
MUL	AB	Multiply A and B	1	1
MUL	Rm,Rm †	Multiply byte register with byte register	3	2
MUL	WRj,WRj †	Multiply word register with word register	3	2

† New features in the A251 assembler and the MCS 251 architecture

NOP	No Operation	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
NOP		No operation	1	1

ORL	ORL destination, source Logical OR	CY	AC	N	OV	Z
		—	—	X	—	X

Mnemonic		Description	Bytes Binary	Bytes Source †
ORL	A,Rn	OR register to accumulator	1	2
ORL	A,dir8	OR dir byte to accumulator	2	2
ORL	A,@Ri	OR indir RAM to accumulator	1	2
ORL	A,#data8	OR immediate data to accumulator	2	2
ORL	dir,A	OR accumulator to dir byte	2	2
ORL	dir,#data8	OR immediate data to dir byte	3	3
ORL	Rm,Rm †	OR byte register to byte register	3	2
ORL	WRj,WRj †	OR word register to word register	3	2
ORL	Rm,#data8 †	OR 8 bit data to byte register	4	3
ORL	WRj,#data16 †	OR 16 bit data to word register	5	4
ORL	Rm,dir8 †	OR dir address to byte register	4	3
ORL	WRj,dir8 †	OR dir address to word register	4	3
ORL	Rm,dir16 †	OR dir address (64K) to byte register	5	4
ORL	WRj,dir16 †	OR dir address (64K) to word register	5	4
ORL	Rm,@WRj †	OR indir address (64K) to byte register	4	3
ORL	Rm,@DRk †	OR indir address (16M) to byte register	4	3

ORL	ORL destination, source Logical OR for bit variables	CY	AC	N	OV	Z
		X	—	X	—	X

Mnemonic		Description	Bytes Binary	Bytes Source †
ORL	C,bit8	OR direct bit to carry; from BIT space	2	2
ORL	C,bit11 †	OR direct bit to carry; from EBIT space	4	3

A

ORL/	ORL/ destination, source Logical OR with Complement	CY	AC	N	OV	Z
		—	—	X	—	X

Mnemonic		Description	Bytes Binary	Bytes Source †
ORL	C,/bit8	OR complement of direct bit to carry; BIT space	2	2
ORL	C,/bit11 †	OR complement of dir bit to carry; EBIT space	4	3

POP	POP Operand from Stack	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
POP	Rm †	Pop byte register from stack	3	2
POP	WRj †	Pop word register from stack	3	2
POP	DRk †	Pop double word register from stack	3	2
POP	dir8	Pop direct byte from stack	2	2

PUSH	PUSH Operand onto Stack	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
PUSH	Rm †	Push byte register onto stack	3	2
PUSH	WRj †	Push word register onto stack	3	2
PUSH	DRk †	Push double word register onto stack	3	2
PUSH	dir8	Push direct byte onto stack	2	2
PUSH	#data8 †	Push immediate data onto stack	4	3
PUSH	#data16 †	Push immediate data (16 bit) onto stack	5	4

RET	RETURN from Subroutine	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
RET		Return from subroutine	1	1

RETI	RETURN from Interrupt	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
RETI		Return from interrupt	1	1

RL	ROTATE Accumulator Left	CY	AC	N	OV	Z
		—	—	X	—	X

Mnemonic		Description	Bytes Binary	Bytes Source †
RL	A	Rotate accumulator left	1	1

RLC	ROTATE Accumulator Left through the Carry	CY	AC	N	OV	Z
		X	—	X	—	X

Mnemonic		Description	Bytes Binary	Bytes Source †
RLC	A	Rotate accumulator left through the carry	1	1

RR	ROTATE Accumulator Right	CY	AC	N	OV	Z
		—	—	X	—	X

Mnemonic		Description	Bytes Binary	Bytes Source †
RR	A	Rotate accumulator right	1	1

RRC	ROTATE Accumulator Right through the Carry	CY	AC	N	OV	Z
		X	—	X	—	X

Mnemonic		Description	Bytes Binary	Bytes Source †
RRC	A	Rotate accumulator right through the carry	1	1

A

SETB	SET Bit Operand	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
SETB	C	Set carry	1	1
SETB	bit8	Set direct bit from BIT space	2	2
SETB	bit11 †	Set direct bit from EBIT space	5	4

SJMP	Short JUMP	CY	AC	N	OV	Z
		—	—	—	—	—

Mnemonic		Description	Bytes Binary	Bytes Source †
SJMP	rel	Short jump (relative address)	2	2

SLL	SHIFT Register Left	CY	AC	N	OV	Z
		X	—	X	—	X

Mnemonic		Description	Bytes Binary	Bytes Source †
SLL	Rm †	Shift byte register left	3	2
SLL	WRj †	Shift word register left	3	2

SRA	SHIFT Register Right (arithmet.) sign extended	CY	AC	N	OV	Z
		X	—	X	—	X

Mnemonic		Description	Bytes Binary	Bytes Source †
SRA	Rm †	Shift byte register right; sign extended	3	2
SRA	WRj †	Shift word register right; sign extended	3	2

SRL	SHIFT Register Right (logic) zero extended	CY	AC	N	OV	Z
		X	—	X	—	X

Mnemonic		Description	Bytes Binary	Bytes Source †
SRL	Rm †	Shift byte register right; zero extended	3	2

A

SRL	SHIFT Register Right (logic) zero extended	CY X	AC —	N X	OV —	Z X

Mnemonic		Description	Bytes Binary	Bytes Source †
SRL	WRj †	Shift word register right; zero extended	3	2

SUB	SUB destination, source Subtraction	CY X	AC X	N X	OV X	Z X

Mnemonic		Description	Bytes Binary	Bytes Source †
SUB	Rm,Rm †	Subtract byte register from byte register	3	2
SUB	WRj,WRj †	Subtract word register from word register	3	2
SUB	DRk,DRk †	Subtract dword register from dregister	3	2
SUB	Rm,#data †	Subtract 8 bit data from byte register	4	3
SUB	Wrj,#data16 †	Subtract 16 bit data from word register	5	4
SUB	Drk,#data16 †	Subtract 16 bit unsigned data from dword reg.	5	4
SUB	Rm,dir †	Subtract direct address from byte register	4	3
SUB	Wrj,dir †	Subtract direct address from word register	4	3
SUB	Rm,dir16 †	Subtract direct address (64K) from byte register	5	4
SUB	Wrj,dir16 †	Subtract direct address (64K) from word register	5	4
SUB	Rm,@WRj †	Subtract indirect address (64K) from byte reg.	4	3
SUB	Rm,@DRk †	Subtract indirect address (16M) from byte reg.	4	3

SUBB	SUBB destination, source Subtraction with Borrow	CY X	AC X	N X	OV X	Z X

Mnemonic		Description	Bytes Binary	Bytes Source †
SUBB	A,Rn	Subtract register from accumulator with borrow	1	2
SUBB	A,dir8	Subtract direct byte from accumulator with borrow	2	2
SUBB	A,@Ri	Subtract indirect byte from accumulator with borrow	1	2
SUBB	A,#data8	Subtract immediate data from accumulator with borrow	2	2

A

SWAP	SWAP Nibbles within the Accumulator	CY —	AC —	N X	OV —	Z X

Mnemonic		Description	Bytes Binary	Bytes Source †
SWAP	A	Swap nibbles within the accumulator	1	1

TRAP	JUMP to the Trap Interrupt	CY —	AC —	N —	OV —	Z —

Mnemonic		Description	Bytes Binary	Bytes Source †
TRAP	Trap †	Jumps to the trap interrupt vector	2	1

XCH	EXCHANGE Operands	CY —	AC —	N —	OV —	Z —

Mnemonic		Description	Bytes Binary	Bytes Source †
XCH	A,Rn	Exchange register with accumulator	2	2
XCH	A,dir8	Exchange direct byte with accumulator	2	2
XCH	A,@Ri	Exchange indirect byte with accumulator	1	2

XCHD	EXCHANGE Digit	CY —	AC —	N —	OV —	Z —

Mnemonic		Description	Bytes Binary	Bytes Source †
XCHD	A,@Ri	Exchange low-order digit in indir. RAM with accumulator	1	2

XRL	EXCL.-OR destination, source Logical Exclusive-OR	CY —	AC —	N X	OV —	Z X

Mnemonic		Description	Bytes Binary	Bytes Source †
XRL	A,Rn	Exclusive-OR register to accumulator	1	2
XRL	A,dir8	Exclusive-OR direct byte to accumulator	2	2
XRL	A,@Ri	Exclusive-OR indirect byte to accumulator	1	2

XRL		EXCL.-OR destination, source Logical Exclusive-OR	CY —	AC —	N X	OV —	Z X

Mnemonic		Description	Bytes Binary	Bytes Source †
XRL	A,#data8	Exclusive-OR immediate data to accumulator	2	2
XRL	dir8,A	Exclusive-OR accumulator to direct byte	2	2
XRL	dir8,#data8	Exclusive-OR immediate data to direct byte	3	3
XRL	Rm,Rm †	Exclusive-OR byte register to byte register	3	2
XRL	WRj,WRj †	Exclusive-OR word register to word register	3	2
XRL	Rm,#data8 †	Exclusive-OR 8 bit data to byte register	4	3
XRL	WRj,#data16 †	Exclusive-OR 16 bit data to word register	5	4
XRL	Rm,dir8 †	Exclusive-OR direct address to byte register	4	3
XRL	WRj,dir8 †	Exclusive-OR direct address to word register	4	3
XRL	Rm,dir16 †	Exclusive-OR direct address (64K) to byte reg.	5	4
XRL	WRj,dir16 †	Exclusive-OR direct address (64K) to word reg.	5	4
XRL	Rm,@WRj †	Exclusive-OR indirect address (64K) to byte reg.	4	3
XRL	Rm,@DRk †	Exclusive-OR indirect address (16M) to byte reg.	4	3

A

MCS 251 Opcode Map

The MCS 251 opcode map is based on the 8051 microcontroller opcode map. It is arranged as two separate maps, one for binary compatible mode, and one for assembly compatible mode. The mode of operation is configurable at reset.

In binary compatible mode the default opcode map is the 8051 microcontroller map with 255 opcodes and one ESCAPE prefix (A5). If one wants to execute a new MCS 251 instruction the opcode must be proceeded with the ESCAPE prefix. This allows the user to take advantage of the new MCS 251 instructions. Unused opcodes in the ESCAPE map are reserved for future use.

At initialization the user may choose to configure the part to take optimum advantage of the new MCS 251 instructions. In this mode the opcode map remains the same except for the register and register indirect instructions of 8051 microcontroller. These instructions, with opcodes with lower nibble between 6 and F, are moved to the ESCAPE map. The new MCS 251 instructions are moved to this freed up space. Unused opcodes are reserved for future use. The displaced 8051 based instructions keep the same machine code (opcode + operand bytes) except that each must now be proceeded by the ESCAPE (A5) prefix. The MCS 251 instructions keep the same machine code, except they no longer need to be proceeded by the ESCAPE (A5) byte.

A

8051 Microcontroller Instructions

Binary Mode	0	1	2	3	4	5	6 - 7	8 - F
Source Mode	0	1	2	3	4	5	A5x6-A5x7	A5x8-A5xF
0	NOP	AJMP adr11	LJMP ADR16	RR A	INC A	INC dir	INC @Ri	INC Rn
1	JBC bit,rel	ACALL adr11	LCALL adr16	RRC A	DEC A	DEC dir	DEC @Ri	DEC Rn
2	JB bit,rel	AJMP adr11	RET	RL A	ADD A,#data	ADD A,dir	ADD A,@Ri	ADD A,Rn
3	JNB bit,rel	ACALL adr11	RETI	RLC A	ADDC A,#data	ADDC A,dir	ADDC A,@Ri	ADDC A,Rn
4	JC rel	AJMP adr11	ORL dir,A	ORL dir,#data	ORL A,#data	ORL A,dir	ORL A,@Ri	ORL A,Rn
5	JNC rel	ACALL adr11	ANL dir,A	ANL dir,#data	ANL A,#data	ANL A,dir	ANL A,@Ri	ANL A,Rn
6	JZ rel	AJMP adr11	XRL dir,A	XRL dir,#data	XRL A,#data	XRL A,dir	XRL A,@Ri	XRL A,Rn
7	JNZ rel	ACALL adr11	ORL c,bit	JMP @A+DPTR	MOV A,#data	MOV dir,#data	MOV @Ri,#data	MOV Rn,#data
8	SJMP rel	AJMP adr11	ANL C,bit	MOVC A,@A+DPTR	DIV AB	MOV dir,dir	MOV dir,@Ri	MOV dir,Rn
9	MOV DPTR,#d16	ACALL adr11	MOV bit,c	MOVC A,@A+DPTR	SUBB A,#data	SUBB A,dir	SUBB A,@Ri	SUBB A,Rn
A	ORL C,/bit	AJMP adr11	MOV C,bit	INC DPTR	MUL AB	ESC	MOV @Ri,dir	MOV Rn,dir
B	ANL C,/bit	ACALL adr11	CPL bit	CPL C	CJNE A,#d8,rel	CJNE A,dir,rel	CJNE @Ri,#d8,rel	CJNE Rn,#d8,rel
C	PUSH dir	AJMP adr11	CLR bit	CLR C	SWAP A	XCH A,dir	XCH A,@Ri	XCH A,Rn
D	POP dir	ACALL adr11	SETB bit	SETB C	DA A	DJNZ dir,rel	XCHD A,@Ri	DJNZ Rn,rel
E	MOVX A,@DPTR	AJMP adr11	MOVX A,@Ri		CLR A	MOV A,dir	MOV A,@Ri	MOV A,Rn
F	MOV @DPTR,A	ACALL adr11	MOVX @Ri,A		CPL A	MOV dir,A	MOV @Ri,A	MOV Rn,A

A

MCS 251 Instructions

Binary Mode	A5x8	A5x9	A5xA	A5xB	A5xC	A5xD	A5xE	A5xF
Source Mode	x8	x9	xA	xB	xC	xD	xE	xF
0	JSLE rel	MOV Rm @WRj+dis	MOVZ WRj,Rm	INC Rm/WRj/ Drk,#short MOV reg,ind			SRA reg	
1	JSG rel	MOV@WRj +dis,Rm	MOVS WRj,Rm	DEC Rm/WRj/ Drk,#short MOV ind,reg			SRL reg	
2	JLE rel	MOV Rm, @DRk+dis			ADD Rm,Rm	ADD WRj,WRj	ADD reg,op2	ADD DRk,DRk
3	JG rel	MOV@DRk +dis,Rm					SLL reg	
4	JSL rel	MOV Wrj, @WRjj+dis			ORL Rm,Rm	ORL WRj,WRj	ORL reg,op2	
5	JSGE rel	MOV@WRj + dis,WRj			ANL Rm,Rm	ANL WRj,WRj	ANL reg,op2	
6	JE rel	MOV Wrj, @DRk+dis			XRL Rm,Rm	XRL WRj,WRj	XRL reg,op2	
7	JNE rel	MOV @Drk +dis,WRj	MOV op1,reg		MOV Rm,Rm	MOV WRj,WRj	MOV reg,op2	MOV DRk,DRk
8		LJMP@WRj EJMP@DRk	EJMP addr24		DIV Rm,Rm	DIV WRj,WRj		
9		LCALL@WR ECALL@DRk	ECALL addr24		SUB Rm,Rm	SUB WRj,WRj	SUB reg,op2	SUB DRk,DRk
A		BIT instructions	ERET		MUL Rm,Rm	MUL WRj,WRj		
B		TRAP			CMP Rm,Rm	CMP WRj,WRj	CMP reg,op2	CMP DRk,DRk
C			PUSH op1					
D			POP op1					
E								
F								

† New features in the A251 assembler and the MCS 251 architecture

Appendix B. Directive Summary

Directive	Format	Description
BIT	*symbol* BIT *bit_address*	Define a bit address in bit data space.
BSEG	BSEG [AT *absolute address*]	Define an absolute segment within the bit address space.
CODE	*symbol* CODE *code_address*	Assign a symbol name to a specific address in the code space.
CSEG	CSEG [AT *absolute address*]	Define an absolute segment within the code address space.
DATA	*symbol* DATA *data_address*	Assign a symbol name to a specific on-chip data address.
DB	[*label*:] DB *expression* [, *expression* ...]	Generate a list of byte values.
DBIT	[*label*:] DBIT *expression*	Reserve a space in bit units.
DD	[*label*:] DD *expression* [, *expression* ...]	Generate a list of double word values.
DS	[*label*:] DS *expression*	Reserve space in byte units.
DSB †	[*label*:] DSB *expression*	Reserve space in byte units.
DSD †	[*label*:] DSD *expression*	Reserve space in double word units.
DSEG	DSEG [AT *absolute address*]	Define an absolute segment within the indirect internal data space.
DSW †	[*label*:] DSW *expression*	Reserve space in word units; advances the location counter of the current segment.
DW	[*label*:] DW *expression* [, *expression* ...]	Generate a list of word values.
END	END	Indicate end of program.
EQU	EQU *expression*	Set symbol value permanently.
EVEN †	EVEN	Ensure word alignment for variables.
EXTRN **EXTERN** †	EXTRN *class* [:*type*] (*symbol* [, *symbol* ...]) EXTERN *class* [:*type*] (*symbol* [,*symbol* ...])	Defines symbols referenced in the current module that are defined in other modules.
IDATA	*symbol* IDATA *idata_address*	Assign a symbol name to a specific indirect internal address.
ISEG	ISEG [AT *absolute address*]	Define an absolute segment within the internal data space.
LABEL †	*name*[:] LABEL [*type*]	Assign a symbol name to a address location within a segment.
LIT †	*symbol* LIT *'literal string'*	Assign a symbol name to a string.
NAME	NAME *modulname*	Specify the name of the current module.
ORG	ORG *expression*	Set the location counter of the current segment.
PROC † **ENDP** †	*name* PROC [*type*] *name* ENDP	Define a function start and end.

† New features in the A251 assembler and the MCS 251 architecture

Directive	Format	Description
PUBLIC	PUBLIC *symbol* [, *symbol* ...]	Identify symbols which can be used outside the current module.
RSEG	RSEG *seg*	Select a relocatable segment.
SEGMENT	*seg* SEGMENT *class* [*reloctype*] [*alloctype*]	Define a relocatable segment.
SET	SET *expression*	Set symbol value temporarily.
USING	USING *expression*	Set the predefined symbolic register address and reserve space for the specified register bank.
XDATA	*symbol* XDATA *xdata_address*	Assign a symbol name to a specific off-chip data address.
XSEG	XSEG [AT *absolute address*]	Define an absolute segment within the external data address space.

Appendix C. Control Summary

Name and Abbreviation	Description
DATA(date) / DA	Places a date string in header (9 characters maximum).
CASE	Enable case sensitive mode for symbol names.
DEBUG / DB	Outputs debug symbol information to object file.
EJECT / EJ ♦	Continue listing on next page.
ERRORPRINT[(*file*)] / EP	Designates a file to receive error messages in addition to the listing.
GEN / GE ♦	Generates a full listing of macro expansions in the listing file.
NOGEN / NOGE ♦	List only the original source text in listing file.
INCLUDE(*file*) / IC ♦	Designates a file to be included as part of the program.
LINK ♦	Place Linker/Locator controls in the Assembler source code.
LIST, NOLIST / LI, NOLI ♦	Print or do not print the assembler source in the listing file.
MODBIN / MB	Select MCS 251 binary mode (default).
MODSRC / MS	Select MCS 251 source mode.
MPL	Enable Macro Processing Language.
NOAMAKE	Disable AutoMAKE information.
NOLINES	Do not generate LINE number information.
NOMACRO / NOMR	Disable Standard Macros
NOMOD51 / NOMO	Do not recognize the 8051-specific predefined special register.
NOMOD251 / NO251	Disable the additional MCS 251 instructions.
NOOBLECT / NOOJ	Designates that no object file will be created.
NOREGISTERBANK/ NORB	Indicates that no banks are used.
NOSYMBOLS / NOSB	No symbol table is listed.
NOSYMLIST,NO SL ♦	Do not list the following symbol definitions in the symbol table.
OBJECT[(*file*)] / OJ	Designate file to receive object code.
PAGELENGTH(*n*) / PL	Sets maximum number of lines in each page of listing file.
PAGEWIDTH(n) / PW	Sets maximum number of characters in each line of listing file.
PRINT[(*file*)] / PR	Designates file to receive source listing.
NOPRINT / NOPR	Designates that no listing file will be created.
REGISTERBANK(*num,...*)	Indicates one or more banks used in program module.
REGUSE ♦	Defines register usage of assembler functions for the C optimizer.
RESTORE / RS ♦	Restores control setting from SAVE stack.
SAVE / SA ♦	Stores current control setting for GEN, LIST and SYMLIST.
SYMLIST, SL ♦	List the following symbol definitions in the symbol table.
TITLE(*string*) / TT	Places a string in all subsequent page headers.
XREF / XR	Creates a cross reference listing of all symbols used in program.

† New features in the A251 assembler and the MCS 251 architecture

♦ — Marks general controls

Directives for Conditional Assembly

Control	Meaning
IF	Translate block if condition is true
ELSE	Translate block if the condition of a previous IF is false.
ELSEIF	Translate block if condition is true and a previous IF or ELSEIF is false.
ENDIF	Marks end of a block.
RESET	Set symbols checked by IF or ELSEIF to false.
SET	Set symbols checked by IF or ELSEIF to true or to a specified value.

C

Appendix D. Macro Summary

This appendix lists the standard macro functions as well as the MPL built-in functions.

Standard Macro Functions

Directive	Description
ENDM	Ends a macro definition.
EXITM	Causes the macro expansion to immediately terminate.
IRP	Specifies a list of arguments to be substituted, one at a time, for a specified parameter in subsequent lines.
IRPC	Specifies an argument to be substituted, one character at a time, for a specified parameter in subsequent lines.
LOCAL	Specifies up to 16 local symbols used within the macro.
MACRO	Begins a macro definition and specifies the name of the macro and any parameters that may be passed to the macro.
REPT	Specifies a repetition factor for subsequent lines in the macro.

D

MPL Built-in Functions.

%'text end-of-line' or %'text'

%(balanced-text)

%*DEFINE(call-pattern)[local-symbol-list](macro-body)

%*DEFINE(macro-name[parameter-list]) [LOCAL local-list] (macro-body)

%n text-n-characters-long

%EQS(arg1,arg2)

%EVAL(expression)

%EXIT

%GES(arg1,arg2)

%GTS(arg1,arg2)

%IF(expression) THEN (balanced-test1) [ELSE (balanced-text2)] FI

%IN

%LEN(balanced-text)

%LES(arg1,arg2)

%LTS(arg1,arg2)

%MATCH(identifier1 delimiter identifier2) (balanced-text)

%METACHAR(balanced-text)

%NES(arg1,arg2)

%OUT(balanced-text)

%REPEAT (expression) (balanced-text)

%SET(macro-id,expression)

%SUBSTR(balanced-text,expression1,expression2)

%WHILE(expression) (balanced-text)

Appendix E. Reserved Symbols

The A251 assembler recognizes a number of predefined or reserved symbols. These are symbols that are reserved by the assembler and may not be redefined in your program. Reserved symbol names include instruction mnemonics, directives, operators, and register names. The following is a list of the symbol names reserved by the A251 assembler.

A
AB
ACALL
ADD
ADDC
AJMP
AND
ANL
AR0
AR1
AR2
AR3
AR4
AR5
AR6
AR7
AT
BIT
BITADDRESSABLE
BLOCK
BSEG
BYTE †
BYTE0 †
BYTE1 †
BYTE2 †
BYTE3 †
C
CALL
CJNE
CLR
CMP
CODE
CONST †
CPL
CSEG
DA
DATA
DB
DBIT
DD †
DEC
DIV
DJNZ
DPTR
DR0 †
DR12 †
DR16 †
DR20 †
DR24 †
DR28 †
DR4 †
DR56 †
DR60 †
DR8 †
DS
DSB †
DSD †
DSEG
DSW †
DW
DWORD †
EBIT †
EBITADDRESSABLE †
ECALL †
ECODE †
EDATA †

† New features in the A251 assembler and the MCS 251 architecture

EJMP †
ELSE
ELSEIF
END
ENDIF
ENDM
ENDP
EQ
EQU
ERET †
EVEN †
EXITM
EXTERN †
EXTRN
FAR †
GE
GT
HCONST †
HDATA †
HIGH
IDATA
IF
INBLOCK
INC
INPAGE
INSEG
IRP
IRPC
ISEG
JB
JBC
JC
JE
JG
JLE
JMP
JNB
JNC
JNE
JNZ
JSG
JSGE
JSL
JSLE
JZ
LABEL †
LCALL
LE
LIT †
LJMP
LOCAL
LOW
LT
MACRO
MOD
MOV
MOVC
MOVH †
MOVS †
MOVX
MOVZ †
MUL
NAME
NCONST †
NE
NEAR †
NOP
NOT
NUL
NUMBER
OFFS †
OR
ORG
ORL
OVERLAYABLE
PAGE
PC
POP
PROC †
PUBLIC
PUSH
R0
R1
R2

† New features in the A251 assembler and the MCS 251 architecture

R3
R4
R5
R6
R7
R8 †
R9 †
R10 †
R11 †
R12 †
R13 †
R14 †
R15 †
REPT
RET
RETI
RL
RLC
RR
RRC
RSEG
SEG
SEGMENT
SET
SETB
SHL
SHR
SJMP
SLL †
SRA †
SRL †
SUB
SUBB
SWAP
TRAP †
UNIT
USING
WORD †
WORD0 †
WORD2 †
WR0 †
WR2 †
WR4 †
WR6 †
WR8 †
WR10 †
WR12 †
WR14 †
WR16 †
WR18 †
WR20 †
WR22 †
WR24 †
WR26 †
WR28 †
WR30 †
XCH
XCHD
XDATA
XOR
XRL
XSEG

In addition to the above symbols the A51 assembler predefines the Special Function Register (SFR) set of the 8051 CPU. This SFR definitions can be disabled with the A51 control **NOMOD51**. The predefined SFR symbols are also reserved symbols and may not be redefined in your program. The following is a list of the SFR names reserved by the A51 assembler when **NOMOD51** is not given.

AC
ACC
B
CY
DPH
DPL
EA
ES
ET0
ET1
EX0
EX1
F0
IE
IE0
IE1
INT0
INT1
IT0
IT1
OV
P
P0
P1
P2
P3
PS
PSW
PT0
PT1
PX0
PX1
RB8
RD
REN
RI
RS0
RS1
RXD
SBUF
SCON
SM0
SM1
SM2
SP
T1
TB8
TCON
TF0
TF1
TH0
TH1
TI
TL0
TL1
TMOD
TO
TR0
TR1
TXD
WR

Appendix F. Listing File Format

This appendix describes the format of the listing file generated by the assembler.

Assembler Listing File Format

The A251 assembler, unless overridden by controls, outputs two files: an object file and a listing file. The object file contains the machine code. The listing file contains a formatted copy of your source code with page headers and, if requested through controls (**SYMBOL** or **XREF**), a symbol table.

Sample A251 Listing

```
A251 MACRO ASSEMBLER  ASAMPLE1                          25/01/95 15:02:23 PAGE     1


DOS MACRO ASSEMBLER A251 Vx.y
OBJECT MODULE PLACED IN ASAMPLE1.OBJ
ASSEMBLER INVOKED BY: F:\RK\ZX\ASM\A251.EXE ASAMPLE1.A51 XREF

LOC    OBJ             LINE     SOURCE

                          1     $NOMOD51
                          2     $INCLUDE (REG52.INC)
                    +1    3  +1 $SAVE
                    +1  106  +1 $RESTORE
                        107
                        108     NAME    SAMPLE
                        109
                        110     EXTRN   CODE    (PUT_CRLF, PUTSTRING)
                        111     PUBLIC  TXTBIT
                        112
------                  113     PROG    SEGMENT CODE
------                  114     PCONST  SEGMENT CODE
------                  115     VAR1    SEGMENT DATA
------                  116     BITVAR  SEGMENT BIT
------                  117     STACK   SEGMENT IDATA
                        118
------                  119             RSEG  STACK
000000                  120             DS    10H  ; 16 Bytes Stack
                        121
000000                  122             CSEG  AT   0
                        123             USING   0  ; Register-Bank 0
                        124     ; Execution starts at address 0 on power-up.
000000 020000     F     125             JMP   START
                        126
------                  127             RSEG  PROG
                        128     ; first set Stack Pointer
000000 758100     F     129     START:  MOV   SP,#STACK-1
                        130
                        131     ; Initialize serial interface
                        132     ; Using TIMER 1 to Generate Baud Rates
                        133     ; Oscillator frequency = 11.059 MHz
000003 758920           134             MOV   TMOD,#00100000B      ;C/T = 0, Mode = 2
000006 758DFD           135             MOV   TH1,#0FDH
000009 D28E             136             SETB  TR1
00000B 759852           137             MOV   SCON,#01010010B
                        138
                        139     ; clear TXTBIT to read form CODE-Memory
```

F

```
00000E C200       F     140             CLR   TXTBIT
                        141
A251 MACRO ASSEMBLER  ASAMPLE1                          25/01/95 15:02:23 PAGE     2

                        142     ; This is the main program. It is a loop,
                        143     ; which displays the a text on the console.
000010                  144     REPEAT:
                        145     ; type message
000010 900000     F     146             MOV   DPTR,#TXT
000013 120000     E     147             CALL  PUTSTRING
000016 120000     E     148             CALL  PUT_CRLF
                        149     ; repeat
000019 8000       F     150             SJMP  REPEAT
                        151     ;
------                  152             RSEG  PCONST
000000 54455354         153     TXT:    DB    'TEST PROGRAM',00H
000004 2050524F
000008 4752414D
00000C 00
                        154
                        155     ; only for demonstration
------                  156             RSEG  VAR1
000000                  157     DUMMY:  DS    21H
                        158
                        159     ; TXTBIT = 0 read text from CODE  Memory
                        160     ; TXTBIT = 1 read text from XDATA Memory
------                  161             RSEG  BITVAR
0000.0                  162     TXTBIT: DBIT  1
                        163
                        164             END



A251 MACRO ASSEMBLER  ASAMPLE1                          25/01/95 15:02:23 PAGE     3

XREF SYMBOL TABLE LISTING
---- ------ ----- -------


N A M E             T Y P E  V A L U E   ATTRIBUTES / REFERENCES

BITVAR . . . . . .  B  SEG   000001H     REL=UNIT, ALN=BIT   116# 161
DUMMY. . . . . .    D  ADDR  000000H R   SEG=VAR1   157#
PCONST . . . . . .  C  SEG   00000DH     REL=UNIT, ALN=BYTE   114# 152
PROG . . . . . .    C  SEG   00001BH     REL=UNIT, ALN=BYTE   113# 127
PUTSTRING. . . . .  C  ADDR  -------     EXT   110# 147
PUT_CRLF . . . . .  C  ADDR  -------     EXT   110# 148
REPEAT . . . . . .  C  ADDR  000010H R   SEG=PROG   144# 150
SAMPLE . . . . . .     108
STACK. . . . . .    I  SEG   000010H     REL=UNIT, ALN=BYTE   117# 119 129
START. . . . . .    C  ADDR  000000H R   SEG=PROG   125 129#
TXT. . . . . .      C  ADDR  000000H R   SEG=PCONST   146 153#
TXTBIT . . . . . .  B  ADDR  0000H.0 R   SEG=BITVAR   111 140 162#
VAR1 . . . . . .    D  SEG   000021H     REL=UNIT, ALN=BYTE   115# 156


REGISTER BANK(S) USED: 0


ASSEMBLY COMPLETE.  0 WARNING(S), 0 ERROR(S)
```

Listing File Heading

Every page has a header on the first line. It contains the words “A251 MACRO ASSEMBLER” followed by the title, if specified. If the title is not specified,

then the module name is used. It is derived from the **NAME** directive (if specified), or from the root of the source filename. On the extreme right side of the header, the date (if specified) and the page number are printed.

In addition to the normal header, the first page of listing includes the A251 listing file header. This header shows the assembler version number, the file name of the object file, if any, and the entire invocation line.

Source Listing

The main body of the listing file is the formatted source listing. A section of formatted source is shown in the following.

Sample Source Listing

```
LOC    OBJ             LINE     SOURCE

000006 758DFD           135             MOV   TH1,#0FDH
```

The format for each line in the listing file depends on the source line that appears on it. Instruction lines contain 4 fields. The name of each field and its meanings is shown in the list below:

- **LOC** shows the location relative or absolute (code address) of the first byte of the instruction. The value is displayed in hexadecimal.
- **OBJ** shows the actual machine code produced by the instruction, displayed in hexadecimal.
- If the object that corresponds to the printed line is to be fixed up (it contains external references or is relocatable), an **F** is printed after the OBJ field. The object fields to be fixed up contain zeros.
- **LINE** shows the **INCLUDE** nesting level, if any, the number of source lines from the top of the program, and the macro nesting level, if any. All values in this field are displayed in decimal numbers.
- **SOURCE** shows the source line as it appears in the file. This line may be extended onto the subsequent lines in the listing file.

DB, **DW**, and **DD** directives are formatted similarly to instruction lines, except the OBJ field shows the data values placed in memory. All data values are shown. If the expression list is long, then it may take several lines in the listing

F

file to display all of the values placed in memory. The extra lines will only contain the LOC and OBJ fields.

The directives that affect the location counter without initializing memory (e.g. **ORG**, **DBIT**, or **DS**) do not use the OBJ field, but the new value of the location counter is shown in the LOC field.

The **SET** and **EQU** directives do not have a LOC or OBJ field. In their place the assembler lists the value that the symbol is set to. If the symbol is defined to equal one of the registers, then REG is placed in this field. The remainder of the directive line is formatted in the same way as the other directives.

Format for Macros, Include Files, and Save Stack

In the listing file, the assembler displays the macro nesting level, the include file level, and the level of the **SAVE**/**RESTORE** stack. These nesting levels are shown before and after the LINE number as shown in the following listing.

```
LOC    OBJ         LINE     SOURCE

                     1     $GEN   ; Enable Macro Listing
                     2
                     3     MYMACRO MACRO         ; A sample macro
                     4             INC     A     ; Macro Level 1
                     5             MACRO2
                     6             ENDM
                     7
                     8     MACRO2  MACRO         ; Macro 2
                     9             NOP           ; Macro Level 2
                    10             ENDM
                    11
                    12
------               13     MYPROG  SEGMENT CODE
------               14             RSEG    MYPROG
                    15
000000 7400         16             MOV     A,#0
                    17             MYMACRO
000002 04           18+1           INC     A     ; Macro Level 1
                    19+1           MACRO2
000003 00           20+2           NOP           ; Macro Level 2
                    21     $INCLUDE (MYFILE.INC) ; A include file
               +1   22     ; This is a comment   ; Include Level 1
               +1   23             MACRO2
000004 00      +1   24+1           NOP           ; Macro Level 1
000005 7401         25             MOV     A,#1
                    26  +1 $SAVE                 ; Save Directive
                    27  +1         MYMACRO       ; SAVE Level 1
000007 04           28+1+1         INC     A     ; Macro Level 1
                    29+1+1         MACRO2
000008 00           30+2+1         NOP           ; Macro Level 2
                    31  +1 $RESTORE
000009 00           32             NOP
                    33             END
```

Symbol Table

The symbol table is a list of all symbols defined in the program along with the status information about the symbol. Any predefined symbols used will also be listed in the symbol table. If the XREF control is used, the symbol table will contain information about where the symbol was used in the program.

The status information includes a **NAME** field, a **TYPE** field, a **VALUE** field, and an **ATTRIBUTES** field.

The **TYPE** field specifies the type of the symbol: ADDR if it is a memory address, NUMB if it is a pure number (e.g., as defined by EQU), SEG if it is a relocatable segment, and REG if a register. For ADDR and SEG symbols, the segment type is added.

The **VALUE** field shows the value of the symbol when the assembly was completed. For REG symbols, the name of the register is given. For NUMB and ADDR symbols, their absolute value (or if relocatable, their offset) is given, followed by A (absolute) or R (relocatable). For SEG symbols, the segment size is given here. Bit address and size are given by the byte part, a period (.), followed by the bit part. The scope attribute, if any, is PUB (public) or EXT (external). These are given after the VALUE field.

The **ATTRIBUTES** field contains an additional piece of information for some symbols: relocation type for segments, segment name for relocatable symbols.

Example Symbol Table Listing

```
SYMBOL TABLE LISTING
------ ----- -------


N A M E             T Y P E  V A L U E   ATTRIBUTES

BITVAR . . . . . .  B  SEG   000001H     REL=UNIT, ALN=BIT
DUMMY. . . . . . .  D  ADDR  000000H R   SEG=VAR1
PCONST . . . . . .  C  SEG   00000DH     REL=UNIT, ALN=BYTE
PROG . . . . . . .  C  SEG   00001BH     REL=UNIT, ALN=BYTE
PUTSTRING. . . . .  C  ADDR  -------     EXT
PUT_CRLF . . . . .  C  ADDR  -------     EXT
REPEAT . . . . . .  C  ADDR  000010H R   SEG=PROG
SAMPLE . . . . . .
STACK. . . . . . .  I  SEG   000010H     REL=UNIT, ALN=BYTE
START. . . . . . .  C  ADDR  000000H R   SEG=PROG
TXT. . . . . . . .  C  ADDR  000000H R   SEG=PCONST
TXTBIT . . . . . .  B  ADDR  0000H.0 R   SEG=BITVAR
VAR1 . . . . . . .  D  SEG   000021H     REL=UNIT, ALN=BYTE
```

If the **XREF** control is used, then the symbol table listing will also contain all of the line numbers of each line of code that the symbol was used. If the value of

the symbol was changed or defined on a line, then that line will have a hash mark (#) following it. The line numbers are displayed in decimal.

Listing File Trailer

At the end of the listing, the assembler prints a message in the following format:

```
REGISTER BANK(S) USED: [r r r r]

ASSEMBLY COMPLETE.  (n) WARNING(S),  (m) ERROR(S)
```

where

r	are the numbers of the register banks used.
n	is the number of warnings found in the program.
m	is the number of errors found in the program.

F

Appendix G. Program Template

The following code template contains guidelines and hints on how to write your own assembly modules. This template, **TEMPLATE.A51**, is stored in the **\C51\ASM** subdirectory.

TEMPLATE.A51

```
$NOMOD51                ; disable predefined 8051 registers
$INCLUDE (REG52.INC)    ; include CPU definition file (for example, 8052)

;------------------------------------------------------------------------------
; Change names in lowercase to suit your needs.
;
; This assembly template gives you an idea of how to use the A251/A51
; Assembler.  You are not required to build each module this way-this is only
; an example.
;
; All entries except the END statement at the End Of File are optional.
;
; If you use this template, make sure you remove any unused segment declarations,
; as well as unused variable space and assembly instructions.
;
; This file cannot provide for every possible use of the A251/A51 Assembler.
; Refer to the A51/A251 User's Guide for more information.
;------------------------------------------------------------------------------

;------------------------------------------------------------------------------
; Module name (optional)
;------------------------------------------------------------------------------
NAME            module_name

;------------------------------------------------------------------------------
; Here, you may import symbols form other modules.
;------------------------------------------------------------------------------
EXTRN   CODE   (code_symbol)    ; May be a subroutine entry declared in
                                ; CODE segments or with CODE directive.

EXTRN   DATA   (data_symbol)    ; May be any symbol declared in DATA segments
                                ; segments or with DATA directive.

EXTRN   BIT    (bit_symbol)     ; May be any symbol declared in BIT segments
                                ; or with BIT directive.

EXTRN   XDATA  (xdata_symbol)   ; May be any symbol declared in XDATA segments
                                ; or with XDATA directive.

EXTRN   NUMBER (typeless_symbol); May be any symbol declared with EQU or SET
                                ; directive

;------------------------------------------------------------------------------
; You may include more than one symbol in an EXTRN statement.
;------------------------------------------------------------------------------
EXTRN   CODE (sub_routine1, sub_routine2), DATA (variable_1)

;------------------------------------------------------------------------------
; Force a page break in the listing file.
;------------------------------------------------------------------------------
$EJECT

;------------------------------------------------------------------------------
; Here, you may export symbols to other modules.  You may use up to 256
; PUBLIC symbols in one module.
;------------------------------------------------------------------------------
PUBLIC  data_variable
```

```
PUBLIC  code_entry
PUBLIC  typeless_number
PUBLIC  xdata_variable
PUBLIC  bit_variable

;------------------------------------------------------------------------------
; You may include more than one symbol in a PUBLIC statement.
;------------------------------------------------------------------------------
PUBLIC  data_variable1, code_table, typeless_num1, xdata_variable1

;------------------------------------------------------------------------------
; Put the STACK segment in the main module.
;------------------------------------------------------------------------------
?STACK          SEGMENT IDATA           ; ?STACK goes into IDATA RAM.
                RSEG    ?STACK          ; switch to ?STACK segment.
                DS      5               ; reserve your stack space
                                        ; 5 bytes in this example.


$EJECT

;------------------------------------------------------------------------------
; Put segment and variable declarations here.
;------------------------------------------------------------------------------

;------------------------------------------------------------------------------
; DATA SEGMENT--Reserves space in DATA RAM--Delete this segment if not used.
;------------------------------------------------------------------------------
data_seg_name   SEGMENT DATA            ; segment for DATA RAM.
                RSEG    data_seg_name   ; switch to this data segment
data_variable:  DS      1               ; reserve 1 Bytes for data_variable
data_variable1: DS      2               ; reserve 2 Bytes for data_variable1

;------------------------------------------------------------------------------
; XDATA SEGMENT--Reserves space in XDATA RAM--Delete this segment if not used.
;------------------------------------------------------------------------------
xdata_seg_name  SEGMENT XDATA           ; segment for XDATA RAM
                RSEG    xdata_seg_name  ; switch to this xdata segment
xdata_variable: DS      1               ; reserve 1 Bytes for xdata_variable
xdata_array:    DS      500             ; reserve 500 Bytes for xdata_array

;------------------------------------------------------------------------------
; INPAGE XDATA SEGMENT--Reserves space in XDATA RAM page (page size: 256 Bytes)
; INPAGE segments are useful for @R0 addressing methodes.
; Delete this segment if not used.
;------------------------------------------------------------------------------
page_xdata_seg  SEGMENT XDATA INPAGE    ; INPAGE segment for XDATA RAM
                RSEG    xdata_seg_name  ; switch to this xdata segment
xdata_variable1:DS      1               ; reserve 1 Bytes for xdata_variable1

;------------------------------------------------------------------------------
; ABSOLUTE XDATA SEGMENT--Reserves space in XDATA RAM at absolute addresses.
; ABSOLUTE segments are useful for memory mapped I/O.
; Delete this segment if not used.
;------------------------------------------------------------------------------
                XSEG    AT 8000H        ; switch absolute XDATA segment @ 8000H
XIO:            DS      1               ; reserve 1 Bytes for XIO port
XCONFIG:        DS      1               ; reserve 1 Bytes for XCONFIG port

;------------------------------------------------------------------------------
; BIT SEGMENT--Reserves space in BIT RAM--Delete segment if not used.
;------------------------------------------------------------------------------
bit_seg_name    SEGMENT BIT             ; segment for BIT RAM.
                RSEG    bit_seg_name    ; switch to this bit segment
bit_variable:   DBIT    1               ; reserve 1 Bit for bit_variable
bit_variable1:  DBIT    4               ; reserve 4 Bits for bit_variable1

;------------------------------------------------------------------------------
; Add constant (typeless) numbers here.
;------------------------------------------------------------------------------
typeless_number EQU     0DH                     ; assign 0D hex
```

```
typeless_num1   EQU     typeless_number-8       ; evaluate typeless_num1

$EJECT

;------------------------------------------------------------------------------
; Provide an LJMP to start at the reset address (address 0) in the main module.
; You may use this style for interrupt service routines.
;------------------------------------------------------------------------------
                CSEG    AT      0       ; absolute Segment at Address 0
                LJMP    start           ; reset location (jump to start)

;------------------------------------------------------------------------------
; CODE SEGMENT--Reserves space in CODE ROM for assembler instructions.
;------------------------------------------------------------------------------
code_seg_name   SEGMENT CODE

                RSEG    code_seg_name   ; switch to this code segment

                USING   0               ; state register_bank used
                                        ; for the following program code.

start:          MOV     SP,#?STACK-1    ; assign stack at beginning

;------------------------------------------------------------------------------
; Insert your assembly program here.  Note, the code below is non-functional.
;------------------------------------------------------------------------------
                ORL     IE,#82H         ; enable interrupt system (timer 0)
                SETB    TR0             ; enable timer 0
repeat_label:   MOV     A,data_symbol
                ADD     A,#typeless_symbol
                CALL    code_symbol
                MOV     DPTR,#xdata_symbol
                MOVX    A,@DPTR
                MOV     R1,A
                PUSH    AR1
                CALL    sub_routine1
                POP     AR1
                ADD     A,R1
                JMP     repeat_label

code_entry:     CALL    code_symbol
                RET

code_table:     DW      repeat_label
                DW      code_entry
                DB      typeless_number
                DB      0
$EJECT

;------------------------------------------------------------------------------
; To include an interrupt service routins, provide an LJMP to the ISR at the
; interrupt vector address.
;------------------------------------------------------------------------------
                CSEG    AT  0BH         ; 0BH is address for Timer 0 interrupt
                LJMP    timer0int

;------------------------------------------------------------------------------
; Give each interrupt function its own code segment.
;------------------------------------------------------------------------------
int0_code_seg   SEGMENT CODE            ; segment for interrupt function
                RSEG    int0_code_seg   ; switch to this code segment
                USING   1               ; register bank for interrupt routine

timer0int:      PUSH    PSW
                MOV     PSW,#08H        ; register bank 1
                PUSH    ACC
                MOV     R1,data_variable
                MOV     DPTR,#xdata_variable
                MOVX    A,@DPTR
                ADD     A,R1
                MOV     data_variable1,A
```

```
                CLR     A
                ADD     A,#0
                MOV     data_variable1+1,A
                POP     ACC
                POP     PSW
                RETI

;-------------------------------------------------------------------------------
; The END directive is ALWAYS required.
;-------------------------------------------------------------------------------
                END             ; End Of File
```

Appendix H. Assembler Differences

This appendix lists the differences between the Intel ASM-51 assembler, the Keil A51 assembler, and the Keil A251 assembler.

Differences Between A51 and A251

Assembly modules written for the A51 assembler may be assembled using the A251 macro assembler. However, since the A251 macro assembler supports the MCS 251 architecture, the following incompatibilities may arise when A51 assembly modules are assembled with the A251 assembler.

- **32-Bit Values in Numeric Evaluations**
  The A51 assembler uses 16-bit values for all numerical expressions. TheA251 macro assembler uses 32-bit values. This may cause problems when overflows occur in numerical expressions. For example:

```
Value     EQU    (8000H + 9000H) / 2
```

  generates the result 800h in A51 since the result of the addition is only a 16-bit value (1000h). However, the A251 assembler calculates a value of 8800h.

- **8051 Pre-defined Special Function Register Symbol Set**
  The default setting of A51 pre-defines the Special Function Register (SFR) set of 8051 CPU. This default SFR set can be disabled with the A51 control **NOMOD51**. A251 does not pre-define the 8051 SFR set. The control **NOMOD51** is accepted by A251 but does not influence any SFR definitions.

- **More Reserved Symbols**
  The A251 macro assembler has more reserved symbols as A51. Therefore it might be necessary to change user-defined symbol names. For example the symbol ECALL cannot be used as label name in A251, since the MCS 251 has a new instruction with that mnemonic.

H

† New features in the A251 assembler and the MCS 251 architecture

- **Object File Differences**
  A251 uses the Intel OMF-251 file format for object files. A51 uses an extended version of the Intel OMF-51 file format. The OMF-51 file format limits the numbers of external symbols and segments to 256 per module. The OMF-251 file format does not have such a limit on the segment and external declarations.

Differences between A51 and ASM51

Assembly modules written for the Intel ASM51 macro assembler can be re-translated with the A51 macro assembler. However you have to take care about the following differences:

- **Enable the MPL Macro Language**
  If your assembly module contains Intel ASM51 macros, the A51 MPL macros need to be enable with the **MPL** control.
- **8051 Pre-defined Interrupt Vectors**
  The Intel ASM51 pre-defines the following symbol names if **MOD51** is active: RESET, EXTI0, EXTI1, SINT, TIMER0, TIMER1. A51 does not pre-define this symbol names.
- **More Reserved Symbols**
  Since the A51 macro assembler supports also conditional assembly and standard macros, A51 has more reserved symbols then Intel ASM51. Therefore it might be necessary to change user-defined symbol names. For example the symbol IF cannot be used as label name in A51, since it is a directive for conditional assembly.
- **Object File Differences**
  A51 generates line number information for source level debugging and file dependencies for AutoMAKE. For compatibility to previous A51 versions and to ASM51, the line number information can be disabled with the A51 control **NOLINES**. The AutoMAKE information can be disabled with the A51 control **NOAMAKE**.

H

Differences between A251 and ASM51

Assembly modules written for Intel ASM51 can be re-translated with the A251 macro assembler. However, since the A251 macro assembler supports also the

MCS 251 architecture, the following incompatibilities can arise when ASM51 modules are re-translated with A251.

- **32-Bit Values in Numeric Evaluations**
  A51 uses for all numerical expressions 16-bit numbers whereas the A251 macro assembler uses 32-bit values. This can cause problems when overflows occur in numerical expressions. For example:

```
Value    EQU    (8000H + 9000H) / 2
```

  has the result 800H in A51 since the result of the addition is only a 16-bit value (1000H), whereas the A251 calculates Value as 8800H.

- **8051 Pre-defined Symbols**
  The default setting of Intel ASM51 pre-defines the Special Function Register (SFR) set and symbol names for reset and interrupt vectors of 8051 CPU. This default symbol set can be disabled with the ASM51 control **NOMOD51**. A251 does not pre-define any of the 8051 SFR or interrupt vector symbols. The control **NOMOD51** is accepted by A251 but does not influence any symbol definitions.

- **More Reserved Symbols**
  The A251 macro assembler has more reserved symbols as ASM51. Therefore it might be necessary to change user-defined symbol names. For example the symbol ECALL cannot be used as label name in A251, since the MCS 251 has a new instruction with that mnemonic.

- **Enable the MPL Macro Language**
  If your assembly module contains Intel ASM51 macros, the A251 MPL macros need to be enable with the **MPL** control.

- **Object File Differences**
  A251 uses the Intel OMF-251 file format for object files. A51 uses an extended version of the Intel OMF-51 file format. The OMF-51 file format limits the numbers of external symbols and segments to 256 per module. The OMF-251 file format does not have such a limit on the segment and external declarations.

  A51 generates also line number information for source level debugging and file dependencies for AutoMAKE. For compatibility to previous A51 versions and to ASM51, the line number information can be disabled with the A51 control **NOLINES**. The AutoMAKE information can be disabled with the A51 control **NOAMAKE**.

H

† New features in the A251 assembler and the MCS 251 architecture

Glossary

A251
The command used to assemble programs using the A251 Macro Assembler.

A51
The command used to assemble programs using the A51 Macro Assembler.

argument
The value that is passed to macro or function.

arithmetic types
Data types that are integral, floating-point, or enumerations.

array
A set of elements all of the same data type.

ASCII
American Standard Code for Information Interchange. This is a set of 256 codes used by computers to represent digits, characters, punctuation, and other special symbols. The first 128 characters are standardized. The remaining 128 are defined by the implementation.

basename
The part of the file name that excludes the drive letter, directory name, and file extension. For example, the basename for the file **`C:\SAMPLE\SIO.A51`** is **`SIO`**.

batch file
A text file that contains MS-DOS commands and programs that can be invoked from the command line.

BCD
See Binary-Coded Decimal (BCD)

Binary-Coded Decimal
A system that is used to encode decimal numbers in binary form. In BCD, each decimal digit of a number is encoded as a binary value 4 bits long. A byte can hold 2 BCD digits – one in the upper 4 bits (or nibble) and one in the lower 4 bits (or nibble).

BL51
The command used to link object files and libraries using the 8051 Code Banking Linker/Locator.

C51
The command used to compile programs using the 8051 Optimizing C Cross Compiler.

constant expression
Any expression that evaluates to a constant non-variable value. Constants may include character, integer, enumeration, and floating-point constant values.

DS51
The command used to load and execute the DS51 Debugger/Simulator.

environment table
The memory area used by MS-DOS to store environment variables and their values.

environment variable
A variable stored in the environment table. These variables provide MS-DOS programs with information like where to find include files and library files.

escape sequence
A backslash (‘\’) character followed by a single letter or a combination of digits that specifies a particular character value in strings and character constants.

expression
A combination of any number of operators and operands that produces a constant value.

function
A combination of declarations and statements that can be called by name that perform an operation and/or return a value.

function call
An expression that invokes and possibly passes arguments to a function.

in-circuit emulator (ICE)
A hardware device that aids in debugging embedded software by providing hardware-level single-stepping, tracing, and break-pointing. Some ICEs provide a trace buffer that stores the most recent CPU events.

include file
A text file that is incorporated into a source file using the **$INCLUDE** control.

keyword
A reserved word with a predefined meaning for the assembler.

LIB51
The command used to manipulate 8051 library files using the 8051 Library Manager.

library
A file that stores a number of possibly related object modules. The linker can extract modules from the library to use in building a target object file.

macro
An identifier that represents a series of lines of assembly text that is defined using the **MACRO** control.

memory model
Any of the models that specifies which memory areas are used for function arguments and local variables.

mnemonic
An ASCII string that is used to represent a machine language opcode in an assembly language instruction.

monitor51
An 8051 program that can be loaded into your target CPU to aid in debugging and rapid product development through rapid software downloading.

object
An area of memory that can be examined. Usually used when referring to the memory area associated with a variable or function.

object file
A file, created by the compiler, that contains the program segment information and relocatable machine code.

OH51
The command used to convert absolute object files into other hexadecimal file formats using the Object File Converter.

opcode
Also called operation code. An opcode is the first byte of a machine code instruction and is usually represented as a 2–digit hexadecimal number. The opcode indicates the type of machine language instruction and the type of operation to perform.

operand
A variable or constant that is used in an expression.

operator
A symbol that specifies how to manipulate the operands of an expression; e.g., +, -, *, /.

parameter
The value that is passed to a macro or function.

pointer
A variable that contains the address of another variable, function, or memory area.

relocatable
Able to be moved or relocated. Not containing absolute or fixed addresses.

RTX51 Full
An 8051 Real-Time Executive that provides a multitasking operating system kernel and library of routines for its use.

RTX51 Tiny
A limited version of RTX51.

scope
The sections of a program where an item (function or variable) can be referenced by name. The scope of an item may be limited to file, function, or block.

SFR
An SFR or Special Function Register is a register in the 8051 internal data memory space that is used to read an write to the hardware components of the 8051. This includes the serial port, timers, counters, I/O ports, and other hardware control registers.

source file
A text file containing assembly program code.

Special Function Register
See SFR.

stack
An area of memory, indirectly accessed by a stack pointer, that shrinks and expands dynamically as items are pushed onto the stack and popped off of the stack. Items in the stack are removed on a LIFO (last-in, first-out) basis.

string
An array of characters that is terminated with a null character (**'\0'**).

string literal
A string of characters enclosed within double quotes (**" "**).

TS51
The command used to load and execute the 8051 TS51 Target Debugger.

two's complement
A binary notation that is used to represent both positive and negative numbers. Negative values are created by complementing all bits of a positive value and adding 1.

whitespace character
Characters that are used as delimiters in C programs such as space, tab, newline, etc.

wild card
One of the MS-DOS characters (? or *) that can be used in place of characters in a filename.

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X