C51ASM

User Guide

3710A–MICRO–10/10

Table of Contents

Section 1
Introduction

C51ASM is a two-pass macro assembler for the Atmel® 8051 family of microcontrollers. C51ASM translates a symbolic code text file (assembly language source) into a machine executable object file. During the first pass, the assembler builds a symbol table from the symbols and labels used in the source file, processes conditional statements and expands macros. In the second pass, the assembler maps the source file into machine code and generates the listing file. C51ASM is an advanced absolute assembler and only generates absolute object files. Users who need a relocatable assembler and linker for modular programming should try the open source as8051 assembler distributed with the Small Device C Compiler (SDCC) or one of the commercially available relocatable assemblers.

C51ASM is meant to be compatible with other standard 8051 assemblers that follow the syntax of the original Intel® ASM51 (discontinued). These include assemblers such as the freely available ASM51 from MetaLink™ Corporation or ASEM-51 by W.W. Heinz; and the commercial A51/Ax51 assembler from Keil. Note that some differences will exist between any assembler, such that every source file may not assemble directly in another assembler without some source modifications or tweaks to the command line.

People already familiar with standard 8051 assemblers can continue to Section 2 “Running the Assembler”. If this is your first introduction to 8051 assembly programming, check out Section 3 “C51ASM Assembly Language” first.

1.1 References

- Ax51 User's Guide. Keil - An ARM® Company, 2004 <http://www.keil.com/support/man/docs/a51/default.htm>.
- ASEM-51 User's Manual V1.3. W.W. Heinz, 2002 <http://plit.de/asem-51/contents.htm>.
- 8051 Cross Assembler User's Manual. Metalink Corporation, 1990 <http://www.metaice.com/ASM51/Files/ASM51MAN.pdf>.
- Intel HEX. Wikipedia, 2009 <http://en.wikipedia.org/w/index.php?title=Intel_HEX&oldid=308770754>.
- SREC (file format). Wikipedia, 2009 <http://en.wikipedia.org/w/index.php?title=SREC_(file_format)&oldid=301168678>.
- External Product Specification for the MCS-51 Object Module Format. Intel Corp., 1982 <http://www.keil.com/download/files/omf51.zip>.

Section 2

Running the Assembler

The Atmel® C51ASM is a command line tool for translating assembly source files. The following sections describe its installation on the supported host platforms.

2.1 Installation

2.1.1 Installation under Microsoft Windows

C51ASM for Microsoft Windows® is available as a zipped archive containing the C51ASM.EXE executable and this manual. To manually install C51ASM as a Windows console application, follow these steps:

- Extract the c51asm_win_1-1.zip archive to your harddisk. This will create a folder called C51ASM.
- Copy the C51ASM folder to your desired install location such as C:\ or C:\Programs Files\.
- Add the executable path, <install>\C51ASM\BIN, to your PATH variable:
  - Microsoft Windows® 98 or earlier

    Append it to your PATH statement in the AUTOEXEC.BAT file. Each different directory is separated with a semicolon as shown in the example below (your path my differ):

    ```
    PATH C:\DOS;C:\UTIL;C:\C51ASM\BIN
    ```

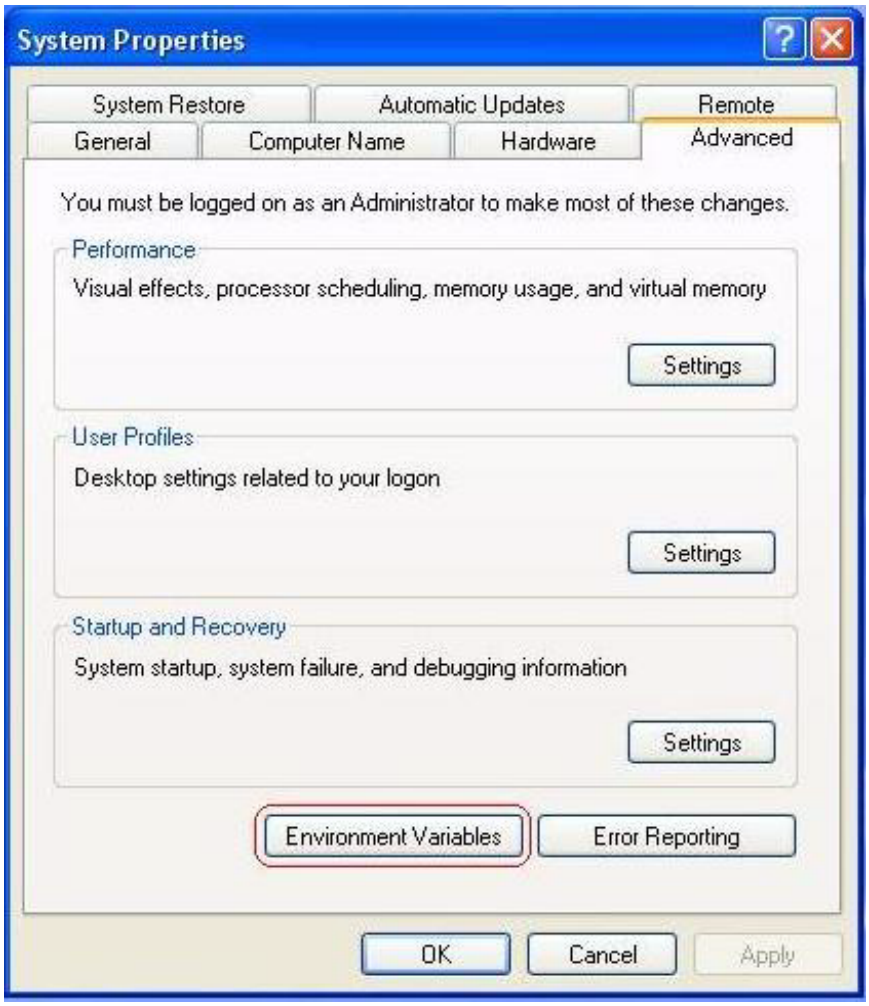
  - Microsoft Windows® 2000, Microsoft Windows® XP or later

    1. From the desktop, right-click **My Computer** and click **Properties** as shown in Figure 2-1.

Figure 2-1. My Computer — Properties

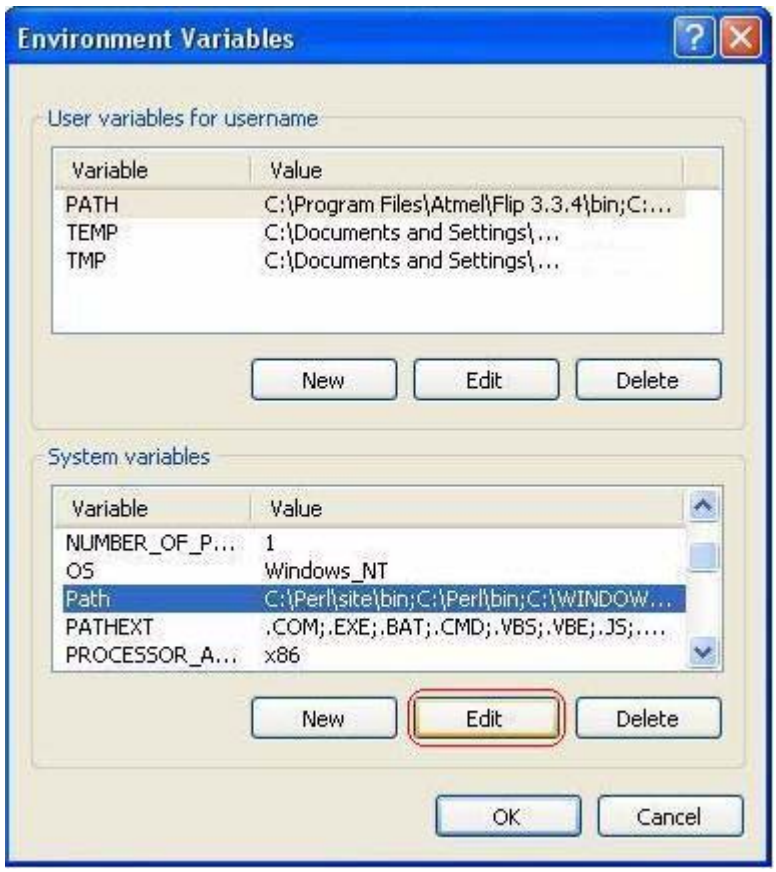
2. In the System Properties window, click on the **Advanced** tab as shown in Figure 2-2.

Figure 2-2. System Properties — Advanced — Environment Variables



3. In the Advanced section, click the **Environment Variables** button as shown in Figure 2-2.
4. In the Environment Variables window, highlight the **Path** variable in the **System variables** section and click **Edit** as shown in Figure 2-3 on page 2-3.

Figure 2-3. Environment Variables — System Variables — Edit Path



5. Append the path to the executable, e.g. <install>\C51ASM\BIN, to the **Path** variable value. Each different directory is separated with a semicolon as shown in Figure 2-4below (your path may differ):

```
Example: C:\Program Files;C:\Winnt;C:\Winnt\System32;C:\C51ASM\BIN
```

Figure 2-4. Edit System Variable

- Optionally define an environment variable C51ASMINC to specify a search path for include files:
  - Microsoft Windows 98 or earlier

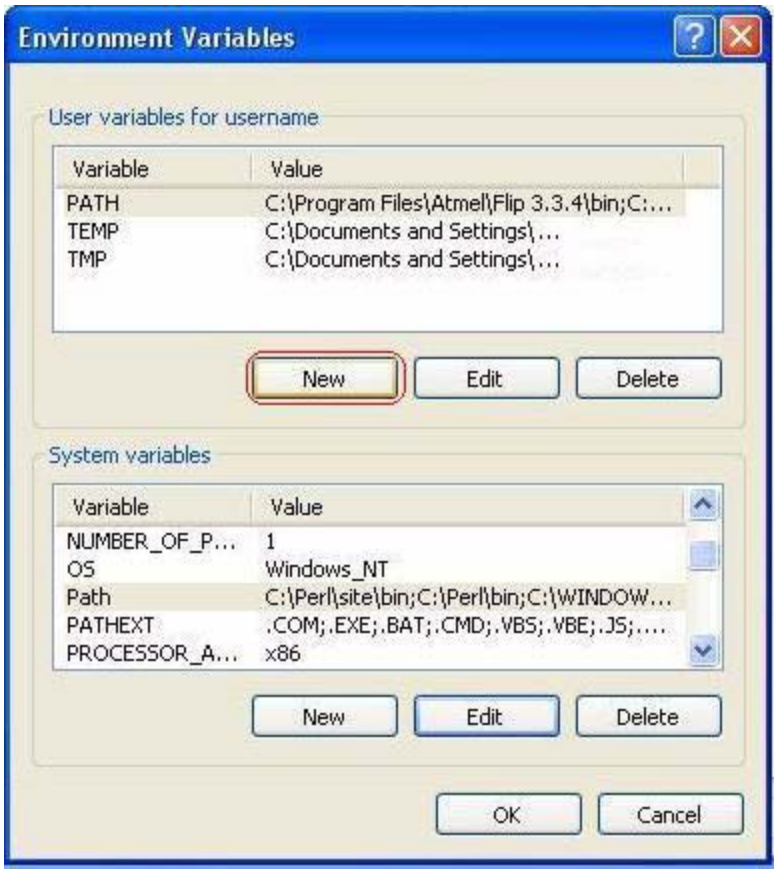
  Set the variable in the AUTOEXEC.BAT file. Each different directory is separated with a semicolon as shown in the example below (your path may differ):

```
SET C51ASMINC=C:\C51ASM\INC;D:\MICROS\MCS51\INCL
```

– Microsoft Windows 2000, Microsoft Windows XP or later

1. From the desktop, right-click **My Computer** and click **Properties**.
2. In the System Properties window, click on the **Advanced** tab.
3. In the Advanced section, click the **Environment Variables** button.
4. In the Environment Variables window, click the **New** button under **User variables**.

Figure 2-5. Environment Variables — User variables — New



5. Finally, in the dialog enter **C51ASMINC** as the variable name. Enter the search path as the variable value and click **OK**. Each different directory is separated with a semicolon as shown in Figure 2-6 below (your path may differ):

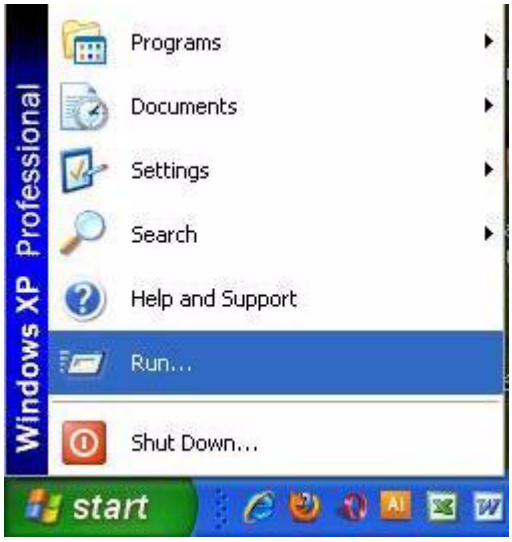
```
Example: C:\C51ASM\INC;D:\PROJECTS\8051\INC
```

Figure 2-6. New User Variable

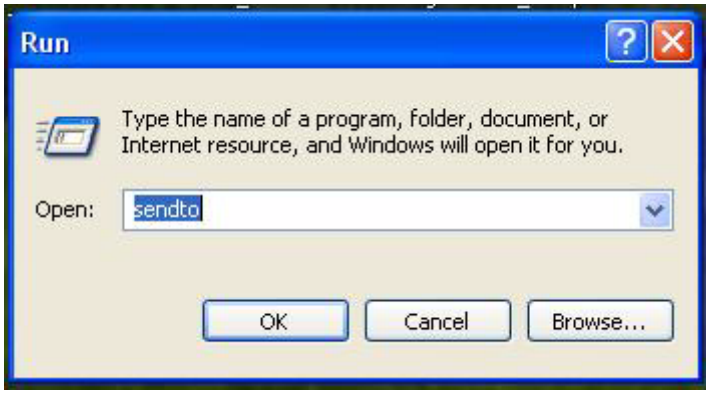
Note that the C51ASMINC variable can be used to point to both the include directory containing header files supplied with C51ASM and any user specified directories. The user can always update the variable to add or remove more items.

- Optionally install the C51ASM.BAT file in your “Send To” menu:

1. From the desktop, click **Start** and then click **Run**.

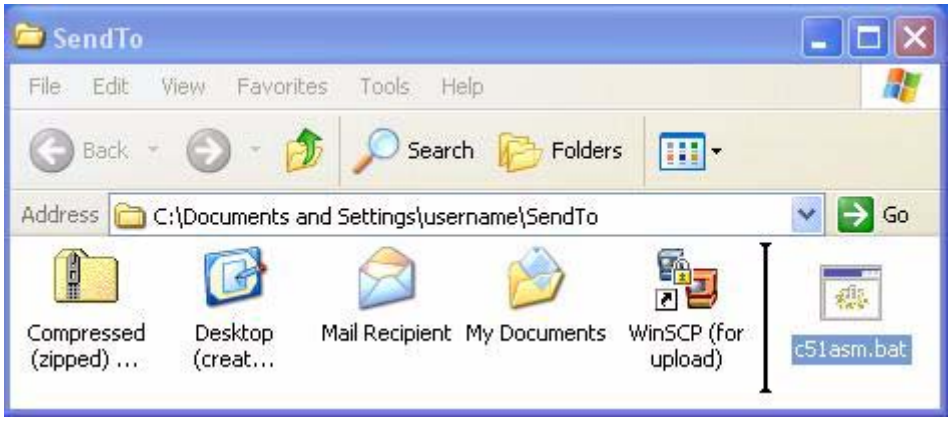


2. In the **Open** box type **sendto** and then click **OK**.



3. Open the directory where you installed C51ASM and copy the C51ASM.BAT file found under DOC to the open **SendTo** window as shown in Figure 2-7. Do NOT create a shortcut as this will not work correctly with a batch file.

Figure 2-7. Copying batch file to SendTo Folder



2.1.2 Installation under Linux

C51ASM for Linux is available as an RPM package or zipped archive.

The RPM package is zipped on ther server. If you have the zipped package, perform the following steps:

1. Extracted the RPM and install

   ```
   unzip c51asm_linux_1-1-rpm.zip
   rpm -i c51asm-1.1.i386.rpm
   ```

   This will install the package under /usr/local/share/c51asm and create symbolic links under /usr/local/bin and /usr/local/share/man/man1.

2. To specify a search path for the include files, define the optional environment variable C51ASMINC.

   For bash, ksh, and sh shells, insert the following line into your .profile file:

   ```
   export C51ASMINC=/usr/local/share/c51asm/include
   ```

   For csh, tcsh, and zsh shells, insert the following line into your .login file:

   ```
   setenv C51ASMINC /usr/local/share/c51asm/include
   ```

3. On most Linux systems, /usr/local/bin and /usr/local/share/man should already be in your PATH and MANPATH variables. If not you will need to add them.

   For bash, ksh, and sh shells, insert the following lines into your .profile file:

   ```
   export PATH=${PATH}:/usr/local/bin
   export MANPATH=${MANPATH}:/usr/local/share/man
   ```

   For csh, tcsh, and zsh shells, insert the following lines into your .login file:

   ```
   setenv PATH ${PATH}:/usr/local/bin
   setenv MANPATH ${MANPATH}:/usr/local/share/man
   ```

If you have the zip archive, you can do a manual install by perform the following steps:

1. Extracted the archive to the desired installation directory

   ```
   unzip c51asm_linux_1-1.zip -d <install-path>
   ```

   or

   ```
   unzip c51asm_linux_1-1.zip
   cp -r c51asm <install-path>
   ```

2. Add <install-path>/c51asm/bin to your PATH.

   For bash, ksh, and sh shells, insert the following line into your .profile file:

   ```
   export PATH=${PATH}:<install-path>/c51asm/bin
   ```

   For csh, tcsh, and zsh shells, insert the following line into your .login file:

   ```
   setenv PATH ${PATH}:<install-path>/c51asm/bin
   ```

3. To specify a search path for include files, define the optional environment variable C51ASMINC.

   For bash, ksh, and sh shells, insert the following line into your .profile file:

   ```
   export C51ASMINC=<install-path>/c51asm/include
   ```

   For csh, tcsh, and zsh shells, insert the following line into your .login file:

   ```
   setenv C51ASMINC <install-path>/c51asm/include
   ```

2.2 Environment

To specify a search path for include files, an optional environment variable C51ASMINC can be defined. The search path can include any number of directories. If C51ASMINC is defined, the assembler searches the specified path(s) for include files that cannot be found in the working directory, nor in the search path specified with the --include option. The path directories will be searched from left to right.

2.2.1 Microsoft Windows Environment

To define the C51ASMINC environment variable under Microsoft Windows, type:

```
SET C51ASMINC=<path>
```

at the Windows console. <path> may be any number of directories separated by ';' characters.

Examples:

```
SET C51ASMINC=C:\C51ASM\INC;D:\MICROS\MCS51\INC
```

If include files cannot be found in the working directory, nor in the --include path (if specified), the assembler searches next C:\C51ASM\INC and finally D:\MICROS\MCS51\INC.

```
SET C51ASMINC=C:\C51ASM\INC;%PATH%
```

If you want to have a default value for C51ASMINC, you can install it as described in Section 2.1.1. You can then create a local setting by following:

```
SET C51ASMINC=<path>;%C51ASMINC%
```

All the directories in <path> will be searched before the any found in the original definition of C51ASMINC.

2.2.2 Linux Environment

To specify a search path for include files, an optional environment variable C51ASMINC can be defined:

- For bash, ksh, and sh shells:

```
export C51ASMINC=<path>
```

- For csh, tcsh, and zsh shells:

```
setenv C51ASMINC <path>
```

<path> may be any number of directories separated by ':' characters. Be sure that the whole definition doesn't contain any blanks or tabs! If C51ASMINC is defined, the assembler searches the specified <path> for include files that can neither be found in the working directory, nor in the search path specified with the --includes option. The <path> directories will be searched from left to right.

Example:

- bash:

```
export C51ASMINC=/usr/local/share/c51asm/include:~/micros/mcs51/inc
```

If include files can neither be found in the working directory, nor in the --includes path (if specified), the assembler searches next /usr/local/share/c51asm/include and finally ~/micros/mcs51/inc.

- csh:

```
setenv C51ASMINC /usr/local/share/c51asm/include
```

If C51ASMINC is defined as above in .login, the assembler finally searches the directory /usr/local/share/c51asm/include for include files.

2.3 Command Line Operation

The assembler is invoked by typing:

```
c51asm [<options>] <source> [<options>]
```

where <source> is the 8051 assembler source and <options> are command line options. Options may appear before or after the source file with the restriction that an option with an optional argument (such as -l) cannot immediately precede the source file when the argument in not specified, or else the source file will be interpreted as the argument. Only the first source file listed will be assembled; other listed files will be ignored.

By default C51ASM will output version and copyright lines, assembly status, a segment usage summary, a register bank summary, and an error summary:

```
C51ASM: advanced C51 macro assembler Version 1.2 (12 Oct. 2010)
Copyright (C) 2010 Atmel Corp.

Pass 1 completed with no warnings and no errors

Pass 2 completed with no warnings and no errors

Segment usage:
Code : 687 bytes
Data : 35 bytes
Idata : 16 bytes
Edata : 0 bytes
Fdata : 0 bytes
Xdata : 5 bytes
Bit : 4 bits

Register banks used: 0, 1

Warnings: 0
Errors: 0
```

Use the --quiet option to turn off all information except warning and error messages.

Errors

When the assembler encounters an error in a source file, a corresponding error message is output to standard error. The error message includes the error code, the name and line number of the source or include file where the error occurred, and the error message itself.

```
ERROR #18 uart.asm(6): Relative offset exceeds -128 / +127.
ERROR #16 uart.asm(23): Divide by zero.
ERROR #2 baud.inc(2): Undefined symbol.
ERROR #6 uart.asm(30): Missing END directive.
```

For more location information use the --columns option to output the column numbers additionally after the line numbers. However, the column number is just a guess and may not be the exact location.

```
ERROR #18 uart.asm(6,17): Relative offset exceeds -128 / +127.
ERROR #16 uart.asm(23,18): Divide by zero.
ERROR #2 baud.inc(2,17): Undefined symbol.
ERROR #6 uart.asm(30,1): Missing END directive.
```

When errors occur in macro expansion lines, the error is flagged with the line number (and optionally column number) of the corresponding line in the macro definition. The macro name is also listed (or REPT for repeat macros).

```
ERROR #3 uart.asm(26,1) TRANSMIT: Duplicate symbol.
ERROR #8 uart.asm(34,3) REPT: Illegal assembly line.
```

When terminating, C51ASM returns an exit code to the calling process:

Table 2-1. Exit Status

Exit Code	Description
0	No Errors
1	Program Errors Detected
2	Fatal Runtime Errors

Note: Warnings are also output on standard error, but do not influence the exit code!

Examples:

```
c51asm program.a51
```

will assemble the 8051 assembly language program program.a51 and produce an Intel-HEX file program.hex.

```
c51asm example.asm -l example.prn
```

will assemble the 8051 assembly language program example.asm and produce an Intel-HEX file example.hex and a listing example.prn.

```
c51asm demo.a51 -o demo.obj -l listing.txt
```

will assemble the 8051 assembly language program demo.a51 and produce an Intel-HEX file demo.obj and a listing listing.txt.

```
c51asm -fM sample.asm -l
```

will assemble the 8051 assembly language program sample.asm and produce a Motorola SREC file sample.srec and a listing sample.lst.

```
c51asm -I /usr/local/include/c51asm:~/8051/inc proto.a51
```

will assemble the program app.a51, while all required include files will be searched first in the default directory, then in /usr/local/include/c51asm, and finally in ~/8051/inc.

```
c51asm --define=USE_AUTOBAUD uart.asm
```

will assemble the program uart.asm, while the symbol USE_AUTOBAUD will be predefined during assembly.

2.3.1 Output Files

C51ASM outputs one or more object files and an optional listing file during compilation. By default, each output file shares the same filename as the source file, but with a different file extension. Output filenames can be changed either on the command line or through assembler controls in the source file. All file names that are specified explicitly on the command line with the --datafile , --listfile , --outfile or --omf-51 options are left unchanged. The arguments for --listfile and --omf-51 are optional. If command line options are not present, the files may be specified by the $DEBUG , $OBJECT and $PRINT controls in the source file.

By default, C51ASM generates an object file in Intel-HEX format. When one of the --debug or --omf-51 options is specified, an absolute OMF-51 module is also generated. C51ASM can generate object files in other formats with the --filetype option as shown below.

Table 2-2. Supported Object Formats

Short	Long	File Format	Extension
-fB	--filetype BINARY --filetype BIN	Binary	.bin
-fG	--filetype GENERIC	Generic Hexadecimal (ASCII)	.mem
-fI	--filetype INTEL --filetype HEX	Intel Hexadecimal (ASCII)	.hex
-fM	--filetype MOTOROLA --filetype SREC	Motorola S-Record (ASCII	.srec
-fO	--filetype OMF51	Intel Object Module Format (Binary)	.omf

2.3.2 Command Line Options

The Atmel C51ASM assembler recognizes the following options listed in Table 2-3. Most command line options have an assembler control equivalent that can be placed directly in the source code in lieu of the command line.

Table 2-3. Command Line Options

Option	Arguments	Short Form	Description
--include	<path1>;...;<pathN> (Win) <path1>:...:<pathN> (Linux)	-I	Specify additional include directories
--define	<symbol>[=<value>[:<type>]]	-D	Define symbol with optional value and type
--datafile	<filename>	-d	Generate data object file called filename
--filetype	<format>	-f<F>	Specify object file format
--debug		-g	Generate debug information in OMF format
--help		-h	Print usage information and exit
--listfile	[<filename>]	-l	Generate listing file optionally called filename
--outfile	<filename>	-o	Generate code object file called filename
--columns		-c	Print column position of errors
--devices			List supported devices and exit
--dptr			Enable dual data pointer alias extensions
--mapfseg	<offset>[:<start>]		Remap FSEG location in object file
--nolist			Suppress listing file
--noobject			Suppress object file(s)
--nopaging			Disable page formatting of listing file
--omf-51	[<filename>]		Generate OMF-51 object optionally called filename

Table 2-3. Command Line Options

Option	Arguments	Short Form	Description
--quiet		-q	Suppress all console output except error messages.
--target	<device>		Use the specified device model
--version			Print version and exit

Note: The short and long options in the same row are equivalent. All option names are case-sensitive!

2.3.2.1 --include

When the --include option is used, the assembler searches the specified path for include files that cannot be found in the working directory. The path may be any number of directories separated by ';' characters on Microsoft Windows or by ':' characters on Linux. The directories will be searched from left to right. Windows allows paths with spaces if the path is listed between quotes. The path specified with the --include option, is searched before the path defined with the (optional) environment variable C51ASMINC!

Syntax:

```
--include <path1>[;<path2>[;<path3> ...]] (Windows)
--include <path1>[:<path2>[:<path3> ...]] (Linux)
```

Short Form:

```
-I <path1>[;<path2>[;<path3> ...]] (Windows)
-I <path1>[:<path2>[:<path3> ...]] (Linux)
```

Example:

```
> c51asm my_prog.asm --include C:\ASM\INC;D:\LIB;"C:\Documents and
Settings\user"
> c51asm my_prog.asm -I /usr/local/lib:~/ASM
```

2.3.2.2 --define

The --define option is useful for selecting particular program variants from the command line that have been implemented with conditional assembly. It allows to define a symbol with a value and a segment type in the command line. Value and type are optional. The segment type of the symbol defaults to NUMBER, if omitted. The symbol value defaults to 0, if omitted. The symbol value may be any numerical constant. The symbol type must be one of the following characters:

Table 2-4. Define Type Codes

Code	Type
B	BIT
C	CODE
D	DATA
I	IDATA
E	EDATA
F	FDATA
X	XDATA
N	NUMBER (default)

Syntax:

```
--define <symbol>[=<value>[:<type>]]
```

Short Form:

```
-D <symbol>[=<value>[:<type>]]
```

Example:

```
> c51asm my_prog.asm --define BAUD=19200
```

2.3.2.3 --filetype

By Default C51ASM generates an object file in Intel Hex format. The --filetype option changes the format of the object file. The following object types are supported.

Table 2-5. Supported Object Formats

Short	Long	File Format	Extension
B	BINARY or BIN	Binary	.bin
G	GENERIC	Generic Hexadecimal (ASCII)	.mem
I	INTEL or HEX	Intel Hexadecimal (ASCII)	.hex
M	MOTOROLA or SREC	Motorola S-Record (ASCII)	.srec
O	OMF51	Intel Object Module Format (Binary)	.omf

Syntax:

```
--filetype <BINARY|BIN|GENERIC|INTEL|HEX|MOTOROLA|SREC|OMF51>
```

Short Form:

```
-f<B|G|I|M|O>
```

Example:

```
> c51asm my_prog.asm --filetype BINARY
> c51asm mu_prog.asm -fM
```

2.3.2.4 --outfile

By Default C51ASM generates a code object file using the source file name with an extension that depends on the file format. The --outfile option explicitly specifies the name of the object file and will override any $OBJECT or $NOOBJECT controls in the assembly source.

Syntax:

```
--outfile <filename>
```

Short Form:

```
-o <filename>
```

Example:

```
> c51asm my_prog.asm --outfile my_prog.hex
> c51asm mu_prog.asm -o my_prog.o
```

2.3.2.5 --datafile

By Default C51ASM generates a data object file using the source file name appended with ".dat" and an extension that depends on the file format. The --datafile option explicitly specifies the name of the data object file. The data object file is only generated if the source contains an FDATA segment with initialized data.

Syntax:

```
--datafile <filename>
```

Short Form:

```
-d <filename>
```

Example:

```
> c51asm my_prog.asm --datafile my_prog.dat.hex
> c51asm mu_prog.asm -d dflash.o
```

2.3.2.6 --listfile

The --listfile option causes C51ASM to generate a listing file, overriding any $NOPRINT controls in the assmebly source. By default the listing file uses the source file name with a ".lst" extension. The optional argument explicitly specifies the name of the listing file and will override any $PRINT controls is the assembly source.

Syntax:

```
--listfile [<filename>]
```

Short Form:

```
-l [<filename>]
```

Example:

```
> c51asm my_prog.asm --listfile my_prog.lst
> c51asm my_prog.asm -l
```

2.3.2.7 --omf-51

The --omf-51 option instructs C51ASM to generate an additional object file in Intel Object Module Format whenever the filetype is not set to OMF51. By default the object file uses the source file name with a ".omf" extension. The optional argument explicitly specifies the name of the object file and will override any $DEBUG controls is the assembly source. The --omf-51 option must be paired with the --debug option or $DEBUG control to include debugging information in the object file.

Syntax:

```
--omf-51 [<filename>]
```

Short Form:

None

Example:

```
> c51asm my_prog.asm --omf-51 my_prog.omf
> c51asm my_prog.asm --omf-51
```

2.3.2.8 --columns

The --columns option tells the assembler to output the column numbers of program errors additionally after the line numbers.

Syntax:

```
--columns
```

Short Form:

```
-c
```

Example:

```
> c51asm my_prog.asm --columns
> c51asm -c my_prog.asm
```

2.3.2.9 --debug

The --debug option instructs C51ASM to generate an object file in Intel Object Module Format and include debugging information such as the symbol table and line number mappings. By default the object file uses the source file name with a ".omf" extension. The object file can be explicitly specified with the --omf-51 option or the $DEBUG control.

Syntax:

```
--debug
```

Short Form:

```
-g
```

Example:

```
> c51asm my_prog.asm --debug
> c51asm -g my_prog.asm
```

2.3.2.10 --devices

The --devices option causes C51ASM to print a list of supported devices. The listed devices can be selected by name with the --target option.

Syntax:

```
--devices
```

Short Form:

None

Example:

```
> c51asm --devices

C51ASM: advanced C51 macro assembler Version 1.1 (12 Oct. 2010)
Copyright (C) 2010 Atmel Corp.

Device name | Flash | Flash | RAM | ERAM  | XRAM
            | Code  | Data  |     |       |
------------+-------+-------+-----+-------+------
(default)   | 65536 | 65536 | 256 | 65536 | Y
```

AT89LP2052	2048	0	256	0	N
AT89LP4052	4096	0	256	0	N
AT89LP213	2048	0	128	0	N
AT89LP214	2048	0	128	0	N
AT89LP216	2048	0	128	0	N
AT89LP428	4096	512	256	512	N
AT89LP52	8192	256	256	0	Y
AT89LP828	8192	1024	256	512	N
AT89LP6440	65536	8192	256	4096	Y
AT89S2051	2048	0	256	0	N
AT89S4051	4096	0	256	0	N
AT89S51	4096	0	128	0	Y
AT89S52	8192	0	256	0	Y
AT89S53	12288	0	256	0	Y
AT89S8252	8192	2048	256	0	Y
AT89S8253	12288	2048	256	0	Y
AT89LS51	4096	0	128	0	Y
AT89LS52	8192	0	256	0	Y
AT89C2051	2048	0	128	0	N
AT89C4051	4096	0	128	0	N
AT89C51	4096	0	128	0	Y
AT89C52	8192	0	256	0	Y
AT89C55WD	20480	0	256	0	Y
AT89C51RC	32768	0	256	256	Y
AT89C51RB2	16384	0	256	1024	Y
AT89C51RC2	32768	0	256	1024	Y
AT89C51RD2	65536	0	256	1792	Y
AT89C51IC2	32768	0	256	1024	Y
AT89C51ID2	65536	2048	256	1792	Y
AT89C51ED2	65536	2048	256	1792	Y

2.3.2.11 --dptr

Some devices include enhanced data pointer modes that modify the behavior of the DPTR-based instructions. The --dptr option enables support for instruction aliases to make the programmer's intent explicit.

Syntax:

```
--dptr
```

Short Form:

None

Example:

```
> c51asm my_prog.asm --dptr
```

2.3.2.12 --mapfseg

Many device programmers map nonvolatile data segments to some location in their input buffers. Use the --mapfseg option to move FSEG data to a different location in the object file. <offset> specifies the address in the object file where the FSEG data will start. <start> optionally specifies the initial address in FSEG that will map to <offset>. If a device is explicitly specified with $DEVICE or --target , <start> should

be limited to either 0 or the normal starting address of FDATA. <offset> and <start> both default to 0. The --mapfseg overrides any $MAPFSEG controls in the assembly source.

Syntax:

```
--mapfseg <offset>[:<start>]
```

Short Form:

None

Example:

```
> c51asm my_prog.asm --mapfseg 2000H:0x200
```

2.3.2.13 --nolist

The --nolist option suppresses generation of the listing file and overrides any $PRINT controls in the assembly source.

Syntax:

```
--nolist
```

Short Form:

None

Example:

```
> c51asm my_prog.asm --nolist
```

2.3.2.14 --noobject

The --noobject option suppresses generation of the object file and overrides any $OBJECT controls in the assembly source. An OMF object will still be generated if requested by the --debug or --omf-51 options.

Syntax:

```
--noobject
```

Short Form:

None

Example:

```
> c51asm my_prog.asm --noobject
```

2.3.2.15 --nopaging

The --nopaging option disabled page formatting of the listing file and overrides any $PAGING controls in the assembly source.

Syntax:

```
--nopaging
```

Short Form:

None

Example:

```
> c51asm my_prog.asm --nopaging
```

2.3.2.16 --quiet

The --quiet option suppresses all generated console output except error messages.

Syntax:

```
--quiet
```

Short Form:

```
-q
```

Example:

```
> c51asm my_prog.asm --quiet
```

2.3.2.17 --target

The --target option selects a specific device model from the list of supported devices. The supported devices can be listed with the --devices option. Some assembler features are only supported for certain devices. The --target option will override any $DEVICE control in the assembly source.

Syntax:

```
--target <device>
```

Short Form:

None

Example:

```
> c51asm my_prog.asm --target at89lp828
```

2.3.2.18 --version

The --version option prints the version of C51ASM.

Syntax:

```
--version
```

Short Form:

None

Example:

```
> c51asm my_prog.asm --version at89lp828
C51ASM: advanced C51 macro assembler Version 1.1 (12 Oct. 2010)
Copyright (C) 2010 Atmel Corp.
```

2.3.2.19 --help

When invoked without parameters, or with the

```
-h
```

or

```
--help
```

options, the assembler displays a help screen listing the available options/

Syntax:

```
--help
```

Short Form:

```
-h
```

Example:

```
> c51asm

C51ASM: advanced C51 macro assembler Version 1.1 (12 Oct. 2010)
Copyright (C) 2010 Atmel Corp.

Usage: c51asm [options] <file> [options]
Options:
-d <datafile> : Generate data object called datafile
-f <B|G|I|M|O> : Specify object file format
-g : Generate debug information in OMF format
-h : This help text
-I <dir> : Specify additional include directories
-l [filename] : Generate list file optionally called filename
-o <filename> : Generate object file called filename
-q : Suppress information messages
-D <symbol>[=<value>[:type]]
: Define symbol optionally with value
More Options:
--columns : Print column location of errors
--datafile <file> : Generate data object called file
--debug : Generate debug information in OMF format
--devices : List supported devices
--dptr : Enable DPTR mode instruction aliases
--help : This help text
--include <dir> : Specify additional include dirs.
--listfile [list] : Create list file optionally called list
--nolist : Disable list file
--noobject : Disable object file
--nopaging : Disable paging in listing file
--omf-51 [file] : Generate OMF object optionally file called file
--outfile <obj> : Generate object file called obj
--quiet : Suppress information messages
--target : Use specified device
--verbose : Enable parser messages
--version : Version information
--filetype <O|G|I|M|B>
: Specify object file format
--define <symbol>[=<value>[:type]]
: Define symbol optionally with value


Report bugs to 8051@atmel.com
```

2.4 Batch File Operation

Under Microsoft Windows® C51ASM may also be run using the included C51ASM.BAT batch file. The batch file allows running of the assembler from the Windows GUI with minimal command shell interaction. The user is free to edit the file to add command line options as needed (See Section 2.3.2 on page 2-10); however, most options have an assembler control equivalent that can be placed directly in the source file. The batch file is located under the <install>\C51ASM\DOC\ folder. There are two methods for running the batch file.

2.4.1 Drag–and–Drop

Using this method you can assemble source files by dragging them on top of the C51ASM.BAT icon.

1. Move or copy C51ASM.BAT to a visible place such as the Desktop or you local project directory.
2. Locate the assembly source file you want to assemble and drag it on top of the C51ASM.BAT icon as shown in Figure 2-8. Note that you may only drag-and-drop one file at a time.

Figure 2-8. Assembling a file by drag and drop on top of batch file

3. A command shell will open with the C51ASM results. Press any key to close the shell.

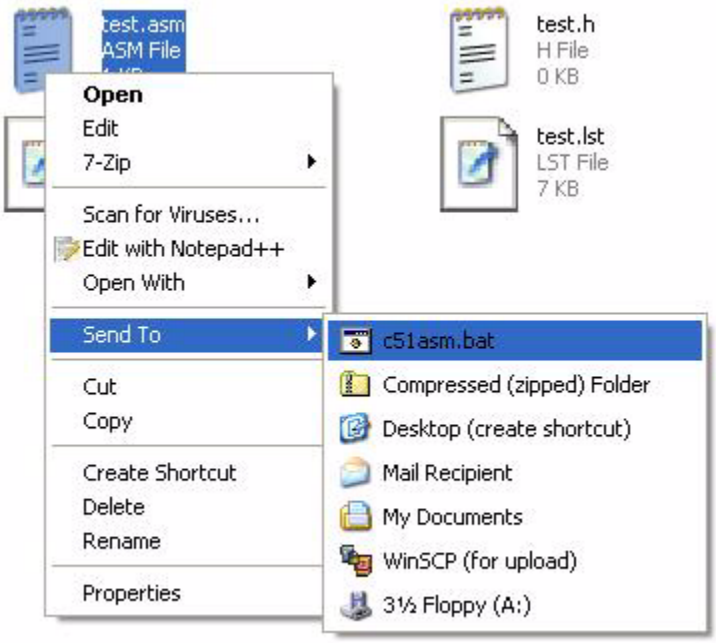
Note that if you want the shell to close automatically, you can remove the “pause” line from the batch file. However, you will lose any diagnostic messages output by the assembler unless you also redirect the results to a log file.

2.4.2 Send To Menu

Using this method you can send source files to the assembler with the right-click context menu.

1. Install C51ASM.BAT in your “Send To” menu as described in Section 2.1.1.
2. Locate the assembly source file you want to assemble.
3. Right-click the source file and select first **Send To** and then **C51ASM.BAT** as shown in Figure 2-9 on page 2-20.
4. A command shell will open with the C51ASM results. Press any key to close the shell.

Figure 2-9. Using the Send To Menu to assemble a file



Section 3

C51ASM Assembly Language

The following section describes the general syntax of assembler statements and the assembler instructions that have been implemented in the Atmel® C51ASM assembler .The user should be familiar with both 8051 microcontrollers and assembly language programming before reading this manual. This manual does not describe the architecture of the 8051 microcontroller family in general, or of the Atmel AT89 devices specifically, nor does it review the basic concepts of assembly language programming.

3.1 Statements

Assembly source files consist of a sequence of statements that include assembler controls, directives or 8051 instructions (mnemonics). Each line of an assembly program can contain only one type of statement. Statements must be contained on exactly one line; multi-line statements are not allowed. Statements have one of the following forms, where everything in [] brackets is optional:

```
[label:] [instruction [arguments]] [;comment]
[label:] directive [arguments] [;comment]
$control [(argument)] [control [(argument)]] [;comment]
```

The maximum length of assembly source lines is not limited; however, lines should be kept shorter than 255 characters for compatibility with other assemblers. The lexical elements of a statement must be separated by spaces or tabs if another delimiting character is not present. The assembler is case insensitive, upper and lower case letters are equivalent, except in character string constants.

Example:

```
LABEL1: MOV A,#0FFH ;define label LABEL1 and load A with FFH
YEAR EQU 2010 ;define symbol for current year
$INCLUDE (AT89LP828.inc) ;include AT89LP828 register definitions header
```

3.2 Comments

Comments are user-defined lines of text that identify and explain the program, but are not processed by the assembler. Comments must be preceded with either a semicolon character (;) or a double forward slash (//). All text after the comment character(s) is ignored by the assembler up to the end of line. Blank lines are also considered to be commentary.

```
;[comment text]
//[comment text]
```

Comments may be included anywhere in your assembly program. A comment can appear on a line by itself or can appear at the end of an instruction. Double semicolon comments (;;) have special meaning inside macro definitions, see Section 4.1.5 ”Macro Operators” on page 4-5.

Example:

```
;This is a comment line
//This is also a comment
NOP ;This is an end comment
NOP //This is a C-style end comment
```

3.3 Symbols

Symbols are user-defined names for addresses, numbers or macros. Symbols are composed of letters (A–Z, a–z), digits (0–9), and the special characters underline (_) and question mark (?). A symbol name can start with any of these characters *except* the digits 0–9. Upper and lower case letters are considered to be equivalent. Assembly language keywords cannot be redefined as user symbols! See “Appendix D - Reserved Keywords” for a list of reserved keywords. C51ASM defines some internal symbols that start with a double question mark (??), see Section 3.3.2 ”Predefined Symbols” on page 3-3. User symbols that start with "??" are not recommended. The following are examples of legal symbols:

```
BUFFER
LOC_1024
Is_this_a_valid_SYMBOL_?
?_?_1
```

Symbols can have unlimited length; however for cross compatibility the maximum significant length is 31 characters. Therefore the symbol should be unique within the first 31 characters.

Symbols are frequently used in assembly programs. A symbolic name is much easier to understand and remember than an address or numeric constant, and a symbol that is defined once and used many times reduces errors. Symbols may be defined in several of ways. The EQU and SET directives define symbols to represent expressions:

```
TEN EQU 10
COUNTER SET R7
```

Labels are special symbols that implicitly define an address of a physical location:

```
LOOP: DJNZ R7, LOOP
```

Symbols can also explicitly define an address:

```
ACC DATA 0E0H
```

3.3.1 Labels

A label is a special symbol that defines a "physical" location (i.e. an address) in your program or data space. Labels must follow all rules that apply to symbol names. When defined, a label must be the first text field in a line. It may be preceded by tabs or spaces and must be immediately followed by a colon character (:) to identify it as a label. Only one label may be defined on a line.

Labels may only be used before statements that have a physical address associated with them. These include instruction mnemonics, data storage directives (DB and DW), and data reservation directives (DS and DBIT). Labels may also be used before empty lines. For example, the following are valid labels:

```
POLL: JNB TI, POLL
TABLE: DB 0,1,2,3,4,5
STACK: DS 16
START:
```

When a label is defined, it receives the current value of the location counter in the currently selected segment and takes the type of that segment. Labels are never typeless.

Once a label has been defined it can be used just like any other symbol. Labels defined in the program space are generally used to transfer program execution to a different location by specifying them as the target of a jump instruction. Data labels are usually used to access variables in memory at the specified location.

3.3.2 Predefined Symbols

C51ASM has a number of predefined (built-in) DATA, BIT and CODE symbols for access to the standrd 80C51 special function registers and interrupt addresses. These predefined symbols can be switched off with the $NOMOD51 control. To suppress the predefined symbols from appearing in the symbol table use the $NOBUILTIN control. For detailed information on predfined symbols and addresses refer to "Appendix C - Predefined Symbols" .

For identification of the assembler and its version number, the following NUMBER type symbols in Table 3-1 are predefined:

Table 3-1. Assembler Info Symbols

Symbol	Value	Description
??C51ASM	8051H	Assembler is C51ASM
??VERSION	0100H	Version is 1.0

Note: These two symbols can not be switched off!

The following device specific symbols in Table 3-2 are also predefined. If a specific device is set by the $DEVICE control or on the command line by --target, their values will reflect the parameters of the selected device.

Table 3-2. Target Device Symbols

Symbol	Default	Description
??DEVICE	0	ID of targeted device
??CODE_SIZE	65536	Code memory size of targeted device
??RAM_SIZE	256	RAM size of targeted device
??ERAM_SIZE	65536	Extended RAM size of targeted device
??FDATA_SIZE	65536	Flash data memory size of targeted device

In addition each supported device listed by the --devices option has a variable defined with a unique value with the following form:

```
??_<devicename>_
```

where <devicename> is replaced by the device name. This allows the currently selected device in ??DEVICE to be checked against specific devices. For example:

```
IF ??DEVICE == ??_AT89LP216_
. . . ;configure for AT89LP216
. . .
ELSEIF ??DEVICE == ??_AT89LP828_
. . . ;configure for AT89LP828
. . .
ENDIF
```

C51ASM reserves the names of several 8051 registers for use as operands. These predefined names are used to access the processor registers as described below in Table 3-3.

Table 3-3. Special Operand Symbols

Symbol	Description
$	The current value of the location counter in the current segment, before a line is assembled. **$** will be the starting address for the assembled line.
A	The 8051 accumulator register. It is used to specify the accumulator as the source or destination operand. Note **A** cannot be used as the address of the accumulator for direct addresses.
AB	The A and B register pair used in MAC, MUL and DIV operations.
AR0–AR7	The absolute data addresses of R0 through R7 in the current register bank. The absolute address for these registers changes depending on the register bank that is currently selected. These symbols are only available when the USING assembler statement is given. Refer to the USING assembler statement for more information on selecting the register bank.
C	The Carry flag. Used by operations that generate a carry or borrow.
DPTR	The selected 16-bit Data Pointer.
PC	The Program Counter. PC can represent the address of the next instruction when used with indexed addressing, i.e A+PC, or the address of the current instruction as a synonym for **$** in expressions. Note PC always refers to the CODE segment.
R0–R7	The eight general purpose 8051 registers in the currently active register bank.

3.4 Constants

C51ASM supports numbers in binary, octal, decimal and hexadecimal format. Numeric constants consist of a sequence of digits followed by a radix specifier. The first character must always be a decimal digit. Alternatively for binary, octal and hexadecimal numbers a radix may be prefixed to the digits. The letter digits and radix designators can be in upper or lower case. The legal digits and radix specifiers are shown in Table 3-4.

Table 3-4. Supported Number Formats

Base	Legal Digits	Radix (Suffix)	Radix (Prefix)
binary	0–1	B	0b
octal	0–7	Q or O	0
decimal	0–9	D or none	
hexadecimal	0–9, A–F, a–f	H	0x

Thus, for example, the following constants are equivalent:

```
 1111111B binary
     177Q octal
     177o octal
      127 decimal
     127d decimal
     07FH hex
0b1111111 binary
     0177 octal
    0x7FH hex
```

Character constants may be used wherever a numeric value is allowed. A character constant consists of one or two printing characters enclosed in single quotes (') or double quotes ("). The single quote character itself can be represented by two subsequent quotes ('). For example:

```
 'A'  8 bit constant: 41H
"b?" 16 bit constant: 623FH
''''  8 bit constant: 27H
```

In DB statements, character constants may have any length to create character strings. For example:

```
DB 'This is sample text!'
```

The difference between single-quoted and double-quoted strings is that double-quoted strings allow C-style escaped characters using the backslash (\). The double quote itself can be represented if it is preceeded by a backslash (\"). Valid escape characters are listed in Table 3-5. Escaped octal and hexidemical codes are not supported.

Table 3-5. Character String Escape Characters

Escape	Value	Description
\0	0x00	Null character
\a	0x07	Alert (bell) character
\b	0x08	Backspace
\f	0x0C	Formfeed
\n	0x0A	Newline
\r	0x0D	Carriage return
\t	0x07	Horizontal tab
\v	0x0B	Vertical tab
\\	0x5C	Backslash
\'	0x27	Single quote
\"	0x22	Double quote

Null terminated strings can be constructed by mixing string and numeric data, or by using the \0.

```
DB 'This is a null terminated string',0
DB "and so is this\0"
```

The location counter may be used to automatically compute the length of a string constant at assembler run time.

```
text: DB "Find the length of this string",0
textlen EQU $-text ; textlen = strlen(text)+1
```

3.5 Expressions

Arithmetic expressions are composed of operands, operators and parentheses. Operands may be user-defined symbols, constants or special assembler symbols. All operands are treated as unsigned 16-bit numbers. Special assembler symbols, that can be used as operands are shown in Table 3-6.

Table 3-6. Special Expression Symbols

Symbol	Description
AR0–AR7	Direct addresses of registers R0 thru R7
$ PC	the location counter of the currently active segment (start address of the current assembler statement)

The following operators in Table 3-7 are implemented:

Table 3-7. Supported Operators

Operator	Text Form	Function
Unary operators:		
+		Identity: +x = x
-		Two's complement: -x = 0-x
~	NOT	One's complement: NOT x = FFFFH-x
	HIGH	High order byte
	LOW	Low order byte
Binary operators:		
+		Unsigned addition
-		Unsigned subtraction
*		Unsigned multiplication
/		Unsigned division
%	MOD	Unsigned remainder
<<	SHL	Logical shift left
>>	SHR	Logical shift right
& or &&	AND	Bitwise Logical and
I or II	OR	Bitwise Logical or
^	XOR	Bitwise Exclusive or
.		Bit operator used for bit-adressable locations
= or ==	EQ	Equal to
<> or !=	NE	Not equal to
<	LT	Less than
<=	LE	Less or equal than
>	GT	Greater than
>=	GE	Greater or equal than

Nonsymbolic keyword operators, such as SHR or AND, must be separated from their operands by at least one blank or tab. Expressions are evaluated from left to right according to operator precedence, which may be overridden by parentheses. Parentheses may be nested to any level. Expressions always evaluate to unsigned 16-bit numbers, while overflows are ignored. When an expression result is to be assigned to an 8-bit quantity, the high byte must be either 00 or FF. For relational operators TRUE evaluates to FFFFH and FALSE evaluates to 0.

Table 3-8. Operator Precedence

Precedence	Operators
1 (highest)	( )
2	NOT HIGH LOW + - (unary)
3	.
4	* / MOD
5	SHL SHR
6	+ - (binary)
7	EQ = NE <> LT < LE <= GT > GE >=
8	AND
9 (lowest)	OR XOR

Example: The expression

```
P1.((87+3)/10 AND -1 SHR 0DH)
```

will evaluate to 91H.

3.6 Segment Type

Every expression is assigned a segment type at assembly time, depending on its operands and operators. The segment type indicates which address space the expression result would point to as an address. There are eight possible segment types as shown in Table 3-9.

Table 3-9. Segment Types

Type	Description
CODE	Address in program memory
DATA	Address in directly addressable data memory
IDATA	Address in indirectly addressable data memory
EDATA	Address in extended data memory
FDATA	Address in flash data memory
XDATA	Address in external data memory
BIT	Address in bit addressable data memory
NUMBER	Generic number (typeless)

Most expression results are typeless and have the segment type NUMBER. However, some expressions will evaluate to a particular segment type when the following six rules are applied:

- Numerical constants are always typeless and are assigned to NUMBER.
- Symbols are assigned a segment type during definition. Symbols that are defined with EQU or SET have NUMBER type. Symbols that are defined with an explicit type directive, e.g. DATA, BIT, etc., will have that type. Labels get the segment type of the segment in which they were defined.
- The result of a unary operation (+, -, NOT, HIGH, LOW) will have the segment type of its operand.
- The results of all binary operations (except "+", "-" and ".") will have no segment type.
- If only one operand in a binary "+" or "-" operation has a segment type, then the result will have that segment type, too. In all other cases, the result will have no segment type.
- The result of the bit operation "." will always have the segment type BIT.

Example:

The following symbols have been defined in a program:

```
OFFSET EQU 16
START CODE 30H
DOIT CODE 0100H
REDLED BIT P1.3
VARIAB4 DATA 20H
PORT DATA 0C8H
RELAY EQU 5
```

- The expression START+OFFSET+3 will have the segment type CODE.
- The expression START+DOIT will be typeless.
- The expression DOIT-REDLED will be typeless.
- The expression 2*VARIAB4 will be typeless.
- The expression PORT.RELAY will have the segment type BIT.

The segment type is checked only when expressions appear as addresses. If the expression result is not typeless and does not have the segment type of the corresponding segment, the instruction is flagged with an error message. The only exceptions are the segment types DATA and IDATA, which are assumed to be compatible in the address range of 0 to 7FH.

Example:

```
Line  I  Addr  Code           Source

   1:          N         30          DSEG AT 030H ;internal RAM
   2:       30 N         01    COUNT: DS 1 ;counter variable
   3:
   4:                                CSEG ;ROM
   5:     0000  C2 30           START: CLR COUNT
                                       ^
                  @@@@@ segment type mismatch @@@@@
```

The CLR instruction is flagged with the error message "segment type mismatch" in the assembler list file, because only a BIT type address is allowed here. However, COUNT is a label with the segment type DATA.

3.7 The 8051 Instruction Set

The Atmel® C51ASM assembler implements all 255 of the standard 8051 machine instructions. A table of all 8051 instructions with their opcodes, mnemonics, arguments, and lengths , along with detailed descriptions can be found in Section 9 “Instruction Set Reference” .

C51ASM also supports the extended instruction set of the Atmel AT89LP family. These instructions may not be supported on every device. The extensions to the instruction set are listed below:

Table 3-10. Instruction Set Extensions

Hex Code	Bytes	Mnemonic	Operands	Description
A5 00	2	BREAK		Software Breakpoint
A5 02	5	EJMP	code addr	Extended Jump
A5 03	2	ASR	M	Arithmetic Shift MAC Accumulator Right
A5 12	5	ECALL	code addr	Extended Subroutine Call
A5 22	2	ERET		Extended Subroutine Return
A5 23	2	LSL	M	Logically Shift MAC Accumulator Left
A5 73	2	JMP	@A+PC	Jump indirect relative to PC
A5 90	4	MOV	/DPTR, #data	Load Alternate Data Pointer with a 16-bit constant
A5 93	2	MOVC	A, @A+/DPTR	Move Code byte relative to Alternate DPTR to Acc
A5 A3	2	INC	/DPTR	Increment Alternate Data Pointer
A5 A4	2	MAC	AB	Multiply and Accumulate {AX,A} and {BX,B}
A5 B6	3	CJNE	A, @R0, code addr	Compare indirect to Acc and jump if not equal
A5 B7	3	CJNE	A, @R1, code addr	Compare indirect to Acc and jump if not equal
A5 E0	2	MOVX	A, @/DPTR	Move External RAM (16-bit addr from Alternate DPTR) to Acc
A5 E4	2	CLR	M	Clear MAC Accumulator
A5 F0	2	MOVX	@/DPTR, A	Move Acc to External RAM (16-bit addr from Alternate DPTR)

In addition to instructions the assembler supports generic jumps and calls that do not represent a specific opcode. The assembler implements the following two generic instructions:

```
JMP <address>
CALL <address>
```

These instructions evaluate to a jump or call, although not necessarily the shortest, that will reach the specified address. JMP may assemble to SJMP, AJMP, LJMP or EJMP, while CALL can evaluate to ACALL, LCALL or ECALL. Note that the assembler decision may not be optimal. For code addresses that are forward references (a location later on in the program), the assembler cannot determine which form is most efficient. For forward references the assembler always generates LJMP or LCALL respectively for standard (<=64KB) mode and EJMP or ECALL respectively for extended (>64KB) mode. However, for backward references this is still a powerful tool to reduce code size without extra trouble.

Some AT89LP devices support special modes for data pointer (DPTR) instructions. C51ASM supports some alias instructions that better capture the intent of the programmer. These aliases are not new instructions, but map to existing instructions. No attempt is made to check if the correct mode has been set when these aliases are used. To enable parsing of the aliases the --dptr option must be included on the command line.

Table 3-11. Data Poiner Aliases

MOVC A, @A+DPTR++	MOVC A, @A+DPTR--	MOVC B, @A+DPTR++	MOVC B, @A+DPTR--
MOVC A, @A+/DPTR++	MOVC A, @A+/DPTR--	MOVC B, @A+/DPTR++	MOVC B, @A+/DPTR--
MOVC A, @DPTR	MOVC A, @/DPTR	MOVC B, @DPTR	MOVC B, @/DPTR
MOVC A, @DPTR++	MOVC A, @DPTR--	MOVC B, @DPTR++	MOVC B, @DPTR--
MOVC A, @/DPTR++	MOVC A, @/DPTR--	MOVC B, @/DPTR++	MOVC B, @/DPTR--
DEC DPTR	DEC /DPTR	MOVX B, @DPTR	MOVX B, @/DPTR
MOVX A, @DPTR++	MOVX A, @DPTR--	MOVX B, @DPTR++	MOVX B, @DPTR--
MOVX A, @/DPTR++	MOVX A, @/DPTR--	MOVX B, @/DPTR++	MOVX B, @/DPTR--
MOVX @DPTR++, A	MOVX @DPTR--, A	MOVX @DPTR++, B	MOVX @DPTR--, B
MOVX @/DPTR++, A	MOVX @/DPTR--, A	MOVX @/DPTR++, B	MOVX @/DPTR--, B

3.8 Assembler Directives

C51ASM provides a number of pseudo instructions for defining symbol values, reserving and initializing storage, and controling the placement of code and data. These statements are not to be confused with processor instructions or assembler controls. They do not produce executable code and, with the exception of the DB, DW, BYTE and WORD statements, 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. For further information see the full reference in Section 5 “Assembler Directives” .

Directives statements may be divided into the following categories:

Table 3-12. Address Control Directives

Directive	Description
ORG	Set the location counter to a specific offset or address.
USING	Specify which register bank to use.

Table 3-13. Segment Control Directives

Directive	Description
BSEG	Define an absolute bit segment, or switch to previous bit segment.
CSEG	Define an absolute code segment, or switch to previous code segment.
DSEG	Define an absolute data segment, or switch to previous data segment.
ESEG	Define an absolute extended data segment, or switch to previous extended data segment.

Table 3-13. Segment Control Directives

Directive	Description
FSEG	Define an absolute flash data segment, or switch to previous flash data segment.
ISEG	Define an absolute indirect data segment, or switch to previous indirect data segment.
XSEG	Define an absolute external data segment, or switch to previous external data segment.

Table 3-14. Segment Defintion Directives

Directive	Description
BIT	Define an address in bit space.
CODE	Define an address in code space.
DATA	Define an address in data space.
DEFINE	Define a symbol value.
EDATA	Define an address in extended data space.
EQU	Define a symbol value.
FDATA	Define an address in flash data space.
IDATA	Define an address in indirect data space.
SET	Set a variable symbol value.
SFR	Define an address in SFR space.
XDATA	Define an address in external data space.

Table 3-15. Conditional Assembly Directives

Directive	Description
IF	Begin an IF-ELSE-ENDIF block.
IFN	Begin an IFN-ELSE-ENDIF block.
IFDEF	Begin an IFDEF-ELSE-ENDIF block.
IFNDEF	Begin an IFNDEF-ELSE-ENDIF block.
IFB	Begin an IFB-ELSE-ENDIF block.
IFNB	Begin an IFNB-ELSE-ENDIF block.
ELSEIF	Begin an alternate IF block.
ELSEIFN	Begin an alternate IFN block.
ELSEIFDEF	Begin an alternate IFDEF block.
ELSEIFNDEF	Begin an alternate IFNDEF block.
ELSEIFB	Begin an alternate IFB block.
ELSEIFNB	Begin an alternate IFNB block.
ELSE	Begin an ELSE block.
ENDIF	Terminate an IF block.

Table 3-16. Macro Defintion Directives

Directive	Description
MACRO	Defines a macro
LOCAL	Declare labels with local scope inside macro
ENDM	End of macro definition
EXITM	Terminates macro expansion immediately
REPT	Defines a block that is expanded a specified number of times

Table 3-17. Memory Initialization Directives

Directive	Description
BYTE	Allocate memory for one or more defined byte values. Alias for DB
DB	Allocate memory for one or more defined byte values.
DW	Allocate memory for one or more defined word values.
WORD	Allocate memory for one or more defined word values. Alias for DW

Table 3-18. Memory Reservation Directives

Directive	Description
DBIT	Reserve space for one or more bits.
DS	Reserve space for one or more bytes.

Table 3-19. Miscellaneous Directives

Directive	Description
END	End of program.
NAME	Define module name

3.9 Assembler Controls

C51ASM implements a number of assembler controls that influence the assembly process and list file generation. There are two groups of controls: primary and general controls.

Primary controls can only be used at the beginning of the program and remain in effect throughout the assembly. They may be preceded only by control statements, blank and commentary lines. If the same primary control is used multiple times with different parameters, the last one counts. Many primary controls also have command line option equivalents. Command line options always override equivalent controls in the assembly source.

General controls may be used everywhere in the program. They perform a single action, or remain in effect until they are cancelled or changed by a subsequent control statement.

A control statement starts with a dollar sign ($) character, followed by one or more assembler controls. Spaces or tabs are allowed between the '$' and the first control. Subsequent controls on the same line must be separated by spaces or tabs. $INCLUDE controls must be the last control on a line. Any controls after the $INCLUDE will be ignored until the end of the line.

Assembler controls may have a number or string type operand, which must always be enclosed in parentheses. Number type operands are decimal, octal or hexadecimal constants in standard C form. String type operands are character strings which are enclosed in parentheses instead of quotes. In analogy to quoted strings, no control characters (including tabs) are allowed within these strings! The string delimiters '(' and ')' are allowed as long as the number of open and closing parentheses is balanced.

If a control statement changes the listing mode, the control statement itself is always listed in the new listing mode!

The following tables lists all the implemented controls and their abbreviations. For further information see the full reference in Section 6 “Assembler Controls” .

Table 3-20. Primary Controls

Control	Abbreviation	Description
$DATE	$DA	inserts date string into page header
$DEBUG	$DB	include debug information into object
$NODEBUG	$NODB	don't include debug information
$DEVICE		select device
$INCDIR		add a directory to search path
$MACRO	$MR	reserve n % of free memory for macros
$NOMACRO	$NOMR	reserve all for the symbol table
$MAPFSEG		remap FSEG location in object file
$MOD51	$MO	enable predefined 80C51 SFR symbols
$MOD52		enable predefined 80C52 SFR symbols
$NOMOD51	$NOMO	disable predefined SFR symbols
$NOBUILTIN		don't list predefined symbols
$NOTABS		don't use tabs in list file
$OBJECT	$OJ	generate object file
$NOOBJECT	$NOOJ	don't generate object file
$PAGING	$PI	enable listing page formatting
$NOPAGING	$NOPI	disable listing page formatting
$PAGELENGTH	$PL	set lines per page for listing
$PAGEWIDTH	$PW	set columns per line for listing
$PRINT	$PR	generate list file
$NOPRINT	$NOPR	don't generate list file
$SYMBOLS	$SB	create symbol table
$NOSYMBOLS	$NOSB	don't create symbol table
$XREF	$XR	create cross reference
$NOXREF	$NOXR	don't create cross reference

Table 3-21. General Controls

Control	Abbreviation	Description
$COND		list full IF .. ENDIF constructions
$NOCOND		don't list lines in false branches
$CONDONLY		list assembled lines only
$EJECT	$EJ	start a new page in list file
$ERROR		force a user-defined error
$MESSAGE		output a message to console
$WARNING		output a warning message to console
$GEN	$GE	list macro calls and expansion lines
$NOGEN	$NOGE	list macro calls only
$GENONLY	$GO	list expansion lines only
$INCLUDE	$IC	include a source file
$LIST	$LI	list subsequent source lines
$NOLIST	$NOLI	don't list subsequent source lines
$SAVE	$SA	save current $LIST/$GEN/$COND state
$RESTORE	$RS	restore old $LIST/$GEN/$COND state
$TITLE	$TT	inserts title string into page header

The default state of the assembler controls corresponds to the following:

```
$DATE( )
$TITLE(Copyright (c) 2010 Atmel Corp.)
$MACRO(50)
$MOD51
$OBJECT NOPRINT NODEBUG
$PAGING PAGELENGTH(64) PAGEWIDTH(132)
$SYMBOLS NOXREF
$COND GEN LIST
```

3.10 Conditional Assembly

C51ASM supports conditional program assembly using several conditional assembly directives. Conditional assembly allows selected parts of source code to be assembled or ignored. Conditional assembly may be used to simplify configuration control and maintenance by implementing different program versions or different memory models within a single source file. Using conditional assembly, you can easily maintain one source module that satisfies several applications. The following directive instructions are supported:

Table 3-22. Conditional Assembly Directives

Directive	Arguments	Description
IF	<expr>	Assemble block if <expr> is true.
IFN	<expr>	Assemble block if <expr> is false.
IFDEF	<symbol>	Assemble block if <symbol> is defined.
IFNDEF	<symbol>	Assemble block if <symbol> is not defined.
IFB	<literal>	Assemble block if <literal> is blank.
IFNB	<literal>	Assemble block if <literal> is not blank.
ELSEIF	<expr>	Assemble block if <expr> is true and a previous IF or ELSEIF is false.
ELSEIFN	<expr>	Assemble block if <expr> is not true and a previous IF or ELSEIF is false.
ELSEIFDEF	<symbol>	Assemble block if <symbol> is defined and a previous IF or ELSEIF is false.
ELSEIFNDEF	<symbol>	Assemble block if <symbol> is not defined and a previous IF or ELSEIF is false.
ELSEIFB	<literal>	Assemble block if <literal> is blank and a previous IF or ELSEIF is false.
ELSEIFNB	<literal>	Assemble block if <literal> is not blank and a previous IF or ELSEIF is false.
ELSE		Assemble block if the condition of a previous IF is false.
ENDIF		Ends an IF block
#IF	<expr>	Assemble block if <expr> is true.
#ELIF	<expr>	Assemble block if <expr> is true and a previous #IF or #ELIF is false.
#IFDEF	<symbol>	Assemble block if <symbol> is defined.
#IFNDEF	<symbol>	Assemble block if <symbol> is not defined.
#ELSE		Assemble block if the condition of a previous #IF is false.
#ENDIF		Ends a #IF block
.IF	<expr>	Assemble block if <expr> is true.
.ELSE		Assemble block if the condition of a previous .IF is false.
.ENDIF		Ends a .IF block

The IF, ELSEIF, ELSE and ENDIF directives define conditional assembly blocks. C51ASM supports a number of variants of the the IF and ELSEIF directives that test different conditions in both positive and negative sense. For this document, assume that IF and ELSEIF are synonymous with their respective variants.

Conditional assembly blocks must start with an IF statement and must conclude with an ENDIF statement. A conditional assembly block can include optional alternative blocks beginning with the ELSEIF and ELSE directives. Each ELSEIF or ELSE statement terminates the previous IF or ELSEIF block. The ENDIF terminates the last ELSEIF or ELSE block and the entire IF-ELSE-ENDIF construct.

Syntax:

```
IF <condition 1>
. . . ;assembled if <condition 1> is TRUE
. . .
ELSEIF <condition 2>
. . . ;assembled if <condition 1> is FALSE,
. . . ;and <condition 2> is TRUE
. . .
ELSEIF <condition n>
. . . ;assembled if <condition 1> through
. . . ;<condition n-1> are FALSE and
. . . ;<condition n> is TRUE
ELSE
. . . ;assembled if <condition 1> through
. . . ;<condition n> are FALSE
ENDIF
```

The body of each conditional and alternative block may contain any number of assembly language statements, including instruction mnemonics, controls, directives, macros and nested conditional IF-ELSE-ENDIF blocks. IF-ELSE-ENDIF blocks may be nested to any depth! Use the $COND, $NOCOND and $CONDONLYcontrols to set the listing mode of conditional assembly blocks.

Example:

Simple IF-ENDIF block checking if NOLISTING is defined.

```
IFDEF NOLISTING
$NOPRINT
ENDIF
```

Example:

IF-ELSE-ENDIF block optimizing for case when X is zero.

```
IF (X == 0)
CLR A
ELSE
MOV A, #X
ENDIF
```

Example:

IF-ELSEIF-ELSE-ENDIF block ensuring macro has valid operands

```
CMP MACRO OP1, OP2
IFB
$ERROR(Missing 1st operand to CMP)
ELSEIFB
$ERROR(Missing 2nd operand to CMP)
ELSE
MOV A, OP1
CLR C
SUBB A, OP2
ENDIF
ENDM
```

Section 4

Macro Processing

The Atmel® C51ASM assembler supports a standand macro processor that is similar to those found in other assemblers. The macro processor is a string replacement tool for writing resuable code blocks (i.e. macros) that are used in one or more places in your assembly source code. Each macro call in a program is replaced by its definition and assembled as if the statements were written explicitly, a process called expansion. Macro expansion occurs at assembly time, so the macro processor can take advantage of the standard assembler facilities. Macro expansion can be suppressed with the $NOMACRO control.

Macros are not subroutines. Subroutines are fixed blocks of code that are defined and assembled once in the program and will generate machine code even if they are never called. Subroutines are called explicitly with the CALL instructions and all the requirements of context-switching must be handled in the program. Macros generate in-line code every time they are called. If a macro is never called, its body will not appear is the assembled program, but if it is called many times the associated code is generated many times as well and the size of the program can increase rapidly. Macros may also generate different code at each call, depending on the state of their parameters. The programmer must determine if common code should be implemented as a subroutine or macro depending on the requirements of the application. The following assembler directives support macro definition:

Table 4-1. Macro Defintion Directives

Directive	Description
MACRO	Defines a macro
LOCAL	Declare labels with local scope inside macro
ENDM	End of macro definition
EXITM	Terminates macro expansion immediately
REPT	Defines a block that is expanded a specified number of times

4.1 Standard Callable Macros

4.1.1 Defining Macros

Standard macros must be defined before they are called in a program. A MACRO directive statement starts a macro definition as follows:

Syntax:

```
<macro-name> MACRO [<parameter 1>[, <parameter 2>[, ... ,<parameter n>]]]
[LOCAL <symbol 1>[,<symbol 2>[, ... , <symbol n>]]]
. . .
<macro-body>
ENDM
```

The macro definition header contains the macro name and formal parameter list. The <macro-name> must be a unique symbol that obeys all the rules of symbol creation. Macro names cannot be keywords and cannot be redefined. The macro can have an optional list of one or more parameters that are passed to the macro body, see “Macro Parameters” on page 4-3.

Following the header is an optional list of symbols declared in local scope using the LOCAL directive, see “Local Symbols” on page 4-4.

The <macro-body> contains the template that will replace the macro name in the assembly source. The body may contain instruction mnemonics, directives, controls, macro calls and even further macro definitions. This is no limit on the number of lines in the body. The macro body and consequently the macro definition is terminated with the ENDM directive. Note that the macro body is treated as text. Any nested macro calls will not be expanded until the parent macro is called. Likewise, any nested macro definitions will not be defined until the parent macro is called. See “Nested and Recursive Macro Calls” on page 4-9 and “Nested Macro Definitions” on page 4-10.

Example:

```
ADD16 MACRO ;definition of ADD16
MOV A, R0 ; add {R1,R0} + {R3,R2}
ADD A, R2 ; result in {B,A}
XCH A, B
CLR A
MOV A, R1
ADDC A, R3
XCH A, B
ENDM
```

4.1.2 Calling Macros

To call a macro, specify the macro name followed by the list of arguments, if any. When a macro is called, the macro body is expanded in place of the calling statement and then assembled as normal source lines. A macro can be called by its name in the subsequent program as often as desired. Note that the macro definition must occur before the macro is called. Forward references to macros are not allowed.

Syntax:

```
<macro-name> [<arg 1>[, <arg 2>[, ... ,<arg n>]]]
```

Example:

```
ADD16 ;call
```

After the call of the macro ADD16, the body lines

```
MOV A, R0 ; add {R1,R0} + {R3,R2}
ADD A, R2 ; result in {B,A}
XCH A, B
CLR A
MOV A, R1
ADDC A, R3
XCH A, B
```

are inserted into the program and assembled.

4.1.3 Macro Parameters

Macro parameters allow callable macros to be called with arguments that are passed to the macro body. This allows for creation of more dynamic, flexible macros that are written once, but can be adapted to each calling context. The names of the formal parameters are specified in the macro definition header following the MACRO keyword, separated by commas. All parameter names must be unique symbols that obey all the rules of symbol creation. Keywords cannot be used as parameter names. Macros may have any number of parameters, as long as they fit on one line. Parameter symbols have local scope within the macro body. The same parameter names may be used in other macros, or as standard symbols in the main assembly code, without conflict.

Parameters in the body of the macro are represented by the parameter name. They may be used any number of times and in any order within the macro body. The assembler cannot recognize a parameter symbol if it is part of another symbol, is contained in a quoted string, or appears in commentary. In these cases the special macro operator '**&**' is required. See "Macro Operators" on page 4-5.

Syntax:

```
<macro-name> MACRO [<parameter 1>[, <parameter 2>[, ... ,<parameter n>]]]
```

Example:

```
macro_with_5_params MACRO p1, p2, p3, p4, p5
```

When a macro is called, arguments can be passed to the macro in place of the parameters. The arguments must be separated by commas. Arguments are non-delimited character strings that, during macro expansion, replace the symbols of the corresponding parameters in the macro body. The first argument replaces the symbol of the first parameter, the second argument replaces the symbol of the second parameter, and so forth, in a process called substitution. Valid macro arguments are:

- Arbitrary sequences of printable characters, not containing spaces, tabs, commas, or semicolons
- Quoted strings (in single or double quotes)
- Single printable characters, preceded by '!' as an escape character
- Character sequences, enclosed in literal brackets < ... >, which may be arbitrary sequences of valid macro arguments (types 1–4), plus spaces, commas and semicolons
- Arbitrary sequences of valid macro arguments (types 1–4)
- Expressions preceded by a '%' character

Example:

```
DELAY MACRO NUM, REGISTER
MOV REGISTER, #NUM
DJNZ REGISTER, $
ENDM

DELAY 16, R7
```

After calling the macro DELAY, the body lines:

```
MOV R7, #16
DJNZ R7, $
```

are inserted into the program, and assembled. The parameter names NUM and REGISTER have been replaced by the macro arguments "16" and "R7".

You may pass a NULL value for a parameter by omitting that parameter from the macro call. If an argument is omitted, the corresponding parameter is replaced by an empty string. Arguments at the end of the call line are omitted by simply passing a fewer number of arguments that parameters. If arguments in the middle of the call are to be omitted, the separating comma is still required. The number of arguments passed to a macro can not be greater than the number of parameters.

Example:

```
MYMACRO MACRO P1,P2,P3,P4,P5,P6,P7,P8
```

The macro MYMACRO has eight parameters. If it is called as follows:

```
MUMACRO A1,,A3,,A5,A6
```

Then the formal parameters P1, P3, P5 and P6 are replaced by the arguments A1, A3, A5 and A6 during substitution. The parameters P2, P4, P7 and P8 are replaced by a zero length string. To recognize empty macro arguments, use the IFB and IFNB conditional assembly directives. (See Section 3.10 ”Conditional Assembly” on page 3-14)

4.1.4 Local Symbols

Macros with internal branches may require labels to be defined within the macro body. However, static label names cause errors when the macro is called more than once because the same label name is defined in multiple places. One way to solve this is to have the macro pass the label name as a parameter. Another way is to generate a label within the macro body. C51ASM hides this from the programmer with the concept of local symbols.

The LOCAL directive declares one or more symbols, separated by commas, to have local scope within a macro. Local symbols must be valid symbols, unique within the macro, and different from the formal parameter names (if any). Keywords cannot be used as local symbol names. If a local symbol has the same name as a global symbol, the local scope takes precedance during substitution. LOCAL statements can only occur directly after a MACRO or REPT statement, before any body lines. Any number of LOCAL statements is allowed.

During macro expansion local macro symbols are replaced with a unique, sequentially-numbered label that increments each time the macro is invoked. These have the format "??xxxx", where xxxx is a unique number in hexadecimal format. To avoid name conflicts with substituted local symbols, is not recommended to use symbols with the format "??xxxx" in the main assembly source.

Example:

The following macro implements compare and jump if greate than or equal:

```
CJGE MACRO OP1, OP2, TARGET
CJNE OP1, OP2, NEQ
SJMP TARGET
NEQ: JNC TARGET
ENDM
```

If the macro CJGE is called multiple times, the label NEQ will be defined multiple times. NEQ can be declared as a local symbol as shown:

```
CJGE MACRO OP1, OP2, TARGET
LOCAL NEQ
CJNE OP1, OP2, NEQ
SJMP TARGET
NEQ: JNC TARGET
ENDM
```

When CJGE is called for the first time, the local symbol NEQ will be replaced by ??0000:

```
CJGE A, #45, somewhere

CJNE A, #45, ??0000
SJMP somewhere
??0000: JNC somewhere
```

when it is called for the second time, NEQ will be replaced by ??0001, and so on:

```
CJGE R3, #56, somewhereelse

CJNE R3, #56, ??0001
SJMP somewhereelse
??0001: JNC somewhereelse
```

4.1.5 Macro Operators

Several special operators are available that are very useful for macro definition, call and expansion:

Table 4-2. Macro Operators

Operator	Description
;;	The macro commentary operator indicates that subsequent text on the line should be ignored.
!	The literal escape operator forces the assmebler to treat the next character literally. The escaped character is passed to the macro, even if it is a control character, while the literal operator itself is removed.
< >	The literal bracket operators pass the entire enclosed character string, including any control characters, to the macro as one argument string, while the outermost pair of brackets is removed.
%	The macro evaluation operator causes the subsequent macro argument to be interpreted as an expression, which will be evaluated before it is passed to the macro.
&	The macro substitution operator separates parameter names from surrounding text, including inside quoted strings and commentary.

4.1.5.1 ;; Operator

The double-semicolon operator (;;) is used to precede comments that are not required in the macro expansion. Single-semicolon (;) and C-style (//) comments are always stored with the macro definition and will appear in the list file. Commentary that begins with (;;) is ignored during macro definition and only appears in the macro definition part of the list file.

Example:

```
DELAY MACRO NUM, REGISTER
MOV REGISTER, #NUM ;; Use a DJNZ loop to delay processor
DJNZ REGISTER, $
ENDM

DELAY 5, R3
```

The commentary should only be visible in the definition of the macro DELAY. When DELAY is called, the macro body is expanded as:

```
MOV R3, #5
DJNZ R3, $
```

4.1.5.2 ! Operator

The exclamation mark operator (!) indicates that a special character is to be passed to a macro literally. This operator enables you to pass as arguments comma (,) and angle bracket characters (< >) that are normally interpreted as delimiters, or a semi-colon (;) that is interpreted as commentary, to a macro. To pass more than one character literally, use the <...> operator.

Example:

```
SEND_ASCII_CHAR MACRO CH
MOV SBUF, #'&CH'
JNB TI, $
CLR TI
ENDM
```

The above macro transmits an ASCII character on the UART. To transmit a special character such as ';' you would call SEND_ASCII_CHAR as follows:

```
SEND_ASCII_CHAR !;
```

which would then be expanded to:

```
MOV SBUF, #';'
JNB TI, $
CLR TI
```

4.1.5.3 <...> Operator

The angle bracket operators (< >) enclose text that should be passed literally to a macro. This operator enables you to pass as arguments comma (,) characters that are normally interpreted as delimiters, or a semi-colon (;) that is interpreted as commentary, to a macro. To pass an angle bracket, use the ! operator.

Example:

```
MAKE_TABLE MACRO NAME, VALUES
NAME: INC A
MOVC A, @A+PC
RET
DB VALUES
ENDM
```

Macro MAKE_TABLE creates a table lookup subroutine with a set of values. To call MAKE_TABLE with any number of values, place the VALUES argument between < > so that it is passed directly to the DB statement.

```
MAKE_TABLE MY_TABLE, <1, 1, 2, 3, 5, 8, 13, 21>
```

will expand to:

```
MY_TABLE: INC A
MOVC A, @A+PC
RET
DB 1, 1, 2, 3, 5, 8, 13, 21
```

4.1.5.4 % Operator

The percent operator (%) causes the subsequent argument to be evaluted as an expression rather than being passed literally. An ASCII representation of the numeric value will be passed to the macro during substitution. The value of the expression must be known on pass 1.

The % operator is frequently used with recursive macros.

Example:

```
SERIES MACRO STARTV, ENDV
IF STARTV LE ENDV
DB STARTV
SERIES STARTV+1, ENDV
ENDIF
ENDM
```

If macro SERIES is called as:

```
SERIES 1,3
```

the expansion would be:

```
DB 1
DB 1+1
DB 1+1+1
```

By prepending the first argument of the recursive call to SERIES with a %, the macro expansion becomes much more readable:

```
SERIES MACRO STARTV, ENDV
IF STARTV LE ENDV
DB STARTV
SERIES %STARTV+1, ENDV
ENDIF
ENDM

SERIES 1,3
DB 1
DB 2
DB 3
```

4.1.5.5 & Operator

The ampersand operator (&) concatenates text and macro parameters by separating parameter names from surrounding text. The & operator is the only way for the macro processor to recognize parameters that are part of a larger symbol, or contained within quotes strings and comments. During substitution, the assembler removes one '&' character from each sequence of '&' characters. To use a & character literally in a macro definition you must write two ampersands '&&'.

Example:

```
MAKE_LABEL MACRO NAM, NUM
NAM&NUM:
ENDM
```

When MAKE_LABEL is called as follows:

```
MAKE_LABEL LABEL, 08
```

the parameters NAM and NUM are substituted during macro expansion as shown below:

```
LABEL08:
```

4.1.6 Exiting Macro Expansion

The EXITM directive immediately terminates macro expansion before the end of the macro body is reached.. When this directive is detected, the macro processor stops expanding the current macro and resumes processing after the next ENDM directive. The EXITM directive only makes sense in conjunction with conditional assembly.

When macro expansion is terminated with EXITM, all IF-ELSE-ENDIF conditional assembly blocks that have been opened within the macro body at the point of the EXITM are closed.

Example:

```
DELAY MACRO NUM, REGISTER
IFB
$ERROR(MACRO DELAY requires two arguments)
EXITM
ELSEIFB
EXITM
ENDIF
MOV REGISTER, #NUM ;; Use a DJNZ loop to delay processor
DJNZ REGISTER, $
ENDM
```

If macro DELAY is called with no arguments like:

```
DELAY
```

It will expand to:

```
$ERROR(MACRO DELAY requires two arguments)
```

4.2 Repeat Macros

A "Repeat" macro is a special built-in macro that expands its body a set number of times. Repeat macros start with the REPT keyword, followed by an <expr> for the number of repetitions, which must be known on pass 1. The repeat macro defintion is terminated with an ENDM directive. Repeat macros don't have a macro name and therefore cannot be called multiple times. They are always expanded immediately after their definition.

Repeat macros can also have local symbols like callable macros.

Syntax:

```
REPT <expr>
[LOCAL <symbol 1>[,<symbol 2>[, ... , <symbol n>]]]
. . .
<macro-body>
. . .
ENDM
```

Example:

```
REPT 5
NOP
ENDM
```

This REPT block will expand to five NOP instructions immediately after its definition:

```
NOP
NOP
NOP
NOP
NOP
```

Repeat macros can be made reusable by nested them in a standard callable macro:

```
DO_NOPS MACRO N
REPT N
NOP
ENDM
ENDM
```

4.3 Nested and Recursive Macro Calls

It is perfectly legal for macro bodies to contain macro calls. If a macro call is encountered during the expansion of a macro, the assembler starts immediately expanding the called macro. The expanded body lines are simply inserted into the expanded macro body of the calling macro, until the called macro is completely expanded. Then the expansion of the calling macro is continued with the body line following the nested macro call. The nested macro is not expanded when the calling macro is defined, only when the calling macro itself is expanded.

Example:

```
LOWER MACRO
MOVC A, @A+DPTR
INC DPTR
ENDM

UPPER MACRO
MOV A,#13
LOWER
MOV @R0,A
INC R0
ENDM
```

The macro UPPER calls the macro LOWER in its body. If UPPER is called the expansion will look like:

```
MOV A,#13
MOVC A, @A+DPTR
INC DPTR
MOV @R0,A
INC R0
```

Macros can also recursively call themselves. There must be a stop condition to prevent the macro from calling itself over and over until the assembler runs out of memory. Conditional assembly can be employed to check the limits of some value within the macro.

Example:

```
SERIES MACRO STARTV, ENDV
IF STARTV LE ENDV
DB STARTV
SERIES %STARTV+1, ENDV
ENDIF
ENDM
```

The macro SERIES defines constants from STARTV through ENDV in ascending order in ROM. If SERIES is called like this:

```
$GENONLY CONDONLY
SERIES 1,7
```

results in the following macro expansion:

```
DB 1
DB 2
DB 3
DB 4
DB 5
DB 6
DB 7
```

4.4 Nested Macro Definitions

It is also valid for a macro body to contain further macro definitions. Nested macro will only be defined when the enclosing macro has been expanded! Therefore, the enclosing macro must have been called before the nested macro can be called, otherwise the macro will be undefined. Nested macro definitions are generally used to define macros with arbitrary names or wrap repeat macros in a reusable macro shell.

Example:

```
MAKE_MACRO MACRO MACNAME, REG
MACNAME MACRO
XCH A, REG
PUSH ACC
XCH A, REG
ENDM
ENDM
```

A call of:

```
MAKE_MACRO PUSH_R0, R0
```

would define the macro:

```
PUSH_RO MACRO
XCH A, R0
PUSH ACC
```

```
XCH A, R0
ENDM
```

while the call:

```
MAKE_MACRO PUSH_AT_R1, @R1
```

would define the following macro:

```
PUSH_AT_R1 MACRO
XCH A, @R1
PUSH ACC
XCH A, @R1
ENDM
```

Example:

In the section on Repeat Macros we introduced the nested repeat block.

```
DO_NOPS MACRO N
REPT N
NOP
ENDM
ENDM
```

The macro call

```
DO_NOPS 4
```

will result in:

```
REPT 4
NOP
ENDM
NOP
NOP
NOP
NOP
```

Section 5

Assembler Directives

In the subsequent paragraphs, all C51ASM assembler directives are described. Lexical symbols are written in lower case letters, while assembler keywords are written in upper case. Instruction arguments are represented by <arg>, <arg1> or something like that. Numeric expressions are represented by <expr>, <expr1> and so on. Syntax elements enclosed in brackets are optional. The ellipsis "..." means always "a list with any number of elements".

Table 5-1. Supported Assembler Directives

Directive	Description
BIT	Define bit address
BSEG	Switch to BIT segment
BYTE	Define bytes (Alias for DB)
CODE	Define CODE address
CSEG	Switch to CODE segment
DATA	Define direct data address
DB	Define bytes
DBIT	Define bits
DEFINE	Define numeric constant
DS	Define space
DSEG	Switch to DATA segment
DW	Define words
EDATA	Define extended RAM address
ELSE	Start an ELSE alternate conditional block
ELSEIF	Start an alternate IF conditional block
ELSEIFB	Start an alternate IFB conditional block
ELSEIFDEF	Start an alternate IFDEF conditional block
ELSEIFN	Start an alternate IFN conditional block
ELSEIFNB	Start an alternate IFNB conditional block
ELSEIFNDEF	Start an alternate IFNDEF conditional block
END	End of program
ENDIF	Terminate an IF conditional block
ENDM	End of macro definition
EQU	Define numeric constant

Table 5-1. Supported Assembler Directives

Directive	Description
ESEG	Switch to EDATA segment
EXITM	Exit macro expansion
FDATA	Define flash data address
FSEG	Switch to FDATA segment
IDATA	Define indirect data address
IF	Start an IF–ELSE–ENDIF conditional block
IFB	Start an IFB–ELSE–ENDIF conditional block
IFDEF	Start an IFDEF–ELSE–ENDIF conditional block
IFN	Start an IFN–ELSE–ENDIF conditional block
IFNB	Start an IFNB–ELSE–ENDIF conditional block
IFNDEF	Start an IFNDEF–ELSE–ENDIF conditional block
ISEG	Switch to IDATA Segment
LOCAL	Define local symbol
MACRO	Define a callable macro
NAME	Define module name
ORG	Origin of segment location
REPT	Define a repeat macro
SET	Define numeric variable
SFR	Define direct SFR address
USING	Using register bank
WORD	Define words (alias for DW)
XDATA	Define external RAM address
XSEG	Switch to XDATA segment

Table 5-2. C Preprocessor Style Directives

Directive	Description
#DEFINE	Define numeric constant
#ELIF	Start an alternate #IF conditional block
#ELSE	Start an #ELSE alternate conditional block
#ENDIF	Terminate an #IF conditional block
#IF	Start an #IF–#ELSE–#ENDIF conditional block
#IFDEF	Start an #IFDEF–#ELSE–#ENDIF conditional block
#IFNDEF	Start an #IFNDEF–#ELSE–#ENDIF conditional block
#UNDEF	Undefine symbol

5.1 DB

Function:

Define bytes in memory

Description:

The DB instruction reserves and initializes a number of bytes with the values defined by the arguments. The arguments may either be expressions (which must evaluate to 8-bit values) or character strings of any length.

DB is only allowed in the CODE and FDATA segments!

Syntax:

```
 DB <arg1> [,<arg2> [,<arg3> ... ]]
.DB <arg1> [,<arg2> [,<arg3> ... ]]
.BYTE <arg1> [,<arg2> [,<arg3> ... ]]
```

Example:

```
DB 19,'January',98,(3*7+12)/11
```

See Also:

– DW

5.2 DW

Function:

Define words in memory

Description:

The DW instruction reserves and initializes a number of words with the values defined by the arguments. Every argument may be an arbitrary expression and requires two bytes of space. DW does not accept character strings with more than two characters.

DW is only allowed in the CODE and FDATA segments!

Syntax:

```
 DW <expr1> [,<expr2> [,<expr3> ... ]]
.DW <expr1> [,<expr2> [,<expr3> ... ]]
.WORD <expr1> [,<expr2> [,<expr3> ... ]]
```

Example:

```
DW 0,0C800H,1999,4711
DW 'AB'
```

See Also:

– DB

5.3 DS

Function:

Allocate byte space in memory

Description:

Reserves a number of uninitialized bytes in the current segment. The value of <expr> must be known on pass 1!

DS is allowed in every segment, except in the BIT segment (use DBIT)!

Syntax:

```
 DS <expr>
.DS <expr>
```

Example:

```
DS 200H
```

See Also:

– DBIT

5.4 DBIT

Function:

Allocate bit space in memory

Description:

Reserves a number of uninitialized bits. The value of <expr> must be known on pass 1!

DBIT is only allowed in the BIT segment! Use DS in other segments.

Syntax:

```
 DBIT <expr>
.DBIT <expr>
```

Example:

```
DBIT 16
```

See Also:

– DS

5.5 DEFINE

Function:

Define numeric constant

Description:

The DEFINE instruction defines a symbol for a numeric constant or a register. If a numeric expression <expr> is assigned to the symbol, it will be of the type NUMBER. If a register <reg> is assigned to the symbol, it will be of the type REGISTER. <reg> may be one of the special assembler symbols A, R0, R1, R2, R3, R3, R4, R5, R6, or R7. Unlike SET, a symbol once defined with DEFINE can never be changed! The values of <expr> and <reg> must be known on pass 1! Forward references are not allowed.

Syntax:

```
#DEFINE <symbol> <expr>
.DEFINE <symbol> <expr>
```

Example:

```
#DEFINE MAXMONTH 12
#DEFINE OCTOBER MAXMONTH-2
#DEFINE COUNTREG R5
.DEFINE TEN 10
.DEFINE FIVE TEN/2
```

See Also:

- EQU
- SET
- #UNDEF

5.6 NAME

Function:

Define module name

Description:

Defines a module name for the OMF-51 object file. If no module name is defined, the module name is derived from the source file name. When generating Intel-HEX file output, the NAME instruction has no effect. The module name must be a legal assembler symbol. Only one NAME instruction is allowed within the program. The symbol however, may be redefined in the subsequent program.

Syntax:

```
 NAME <symbol>
.NAME <symbol>
```

Example:

```
NAME My_1st_Program
```

See Also:

None

5.7 ORG

Function:

Set origin of segment location

Description:

Sets the location counter of the current segment to the value <expr>. The value of <expr> must be known on pass 1! It must be greater or equal to the segment base address. The default value of all location counters at program start is 0.

Syntax:

```
 ORG <expr>
.ORG <expr>
```

Example:

```
ORG 08000H
```

See Also:

- BSEG
- CSEG
- DSEG
- ESEG
- FSEG
- ISEG
- XSEG

5.8 USING

Function:

Use Register Bank

Desciption:

Sets the register bank used to <expr>, which must be in the range of 0...3. The USING instruction only affects the values of the special assembler symbols AR0, ... , AR7 representing the direct addresses of registers R0, ... , R7 in the current register bank. The value of <expr> must be known on pass 1! The default value for the register bank is 0.

Syntax:

```
 USING <expr>
.USING <expr>
```

Example:

```
USING 1
```

See Also:

None

5.9 END

Function:

End of program

Description:

This must be the last statement in the source file. After the END statement only commentary and blank lines are allowed!

Syntax:

```
 END
.END
```

Example:

```
END ;end of program
```

See Also:

None

5.10 EQU

Function:

Define numeric constant

Description:

The EQU instruction defines a symbol for a numeric constant or a register. If a numeric expression <expr> is assigned to the symbol, it will be of the type NUMBER. If a register <reg> is assigned to the symbol, it will be of the type REGISTER. <reg> may be one of the special assembler symbols A, R0, R1, R2, R3, R3, R4, R5, R6, or R7. Unlike SET, a symbol once defined with EQU can never be changed! The values of <expr> and <reg> must be known on pass 1! Forward references are not allowed.

Syntax:

```
<symbol> EQU <expr>
<symbol> EQU <reg>
.EQU <symbol> , <expr>
.EQU <symbol> , <reg>
.EQU <symbol> = <expr>
.EQU <symbol> = <reg>
```

Example:

```
MAXMONTH EQU 12
OCTOBER EQU MAXMONTH-2
COUNTREG EQU R5
.EQU TEN, 10
.EQU FIVE = TEN/2
```

See Also:

– DEFINE, SET, #UNDEF

5.11 CODE

Function:

Define code memory address

Description:

The CODE directive defines symbolic addresses for the program memory segment (address space). The symbol will be of type CODE and cannot be redefined. The value of <expr> must be known on pass 1!

Syntax:

```
<symbol> CODE <expr>
```

Example:

```
EPROM CODE 08000H
```

See Also:

- BIT
- DATA, IDATA
- EDATA, FDATA
- XDATA

5.12 DATA

Function:

Define direct data address

Description:

The DATA directive defines symbolic addresses for the directly addressable internal data memory segment (address space). If the numeric value of the address is between 0 and 127 decimal, it is an address of an internal RAM location. If the numeric value of the address is between 128 and 255, it is an address of a Special Function Register. Addresses greater than 255 are illegal and will be flagged as an error. The symbol will be of type DATA and cannot be redefined. The value of <expr> must be known on pass 1!

Syntax:

```
<symbol> DATA <expr>
```

Example:

```
STACK DATA 7
```

See Also:

- BIT
- CODE
- IDATA
- EDATA, FDATA
- XDATA

5.13 SFR

Function:

Define direct SFR address

Description:

The SFR directive defines symbolic addresses for the directly addressable special function register space in the DATA memory segment. SFR is similar to DATA except that the numeric value of the address must be between 128 and 255. Addresses less than 128 or greater than 255 are illegal and will be flagged as an error. The symbol will be of type DATA and cannot be redefined. The value of <expr> must be known on pass 1!

Syntax:

```
<symbol> SFR <expr>
SFR <symbol> = <expr>
```

Example:

```
P0 SFR 80H
```

See Also:

– DATA

5.14 BIT

Function:

Define bit address

Description:

The BIT directive defines symbolic addresses for the bit addressable segment of internal data memory. If the numeric value of the address is between 0 and 127 decimal, it is a bit address mapped in an internal RAM location. If the numeric value of the address is between 128 and 255, it is an address of a bit located in a Special Function Register. Addresses greater than 255 are illegal and will be flagged as an error. The symbol will be of type BIT and cannot be redefined. The value of <expr> must be known on pass 1!

Syntax:

```
<symbol> BIT <expr>
```

Example:

```
REDLED BIT P1.5
```

See Also:

– CODE
– DATA
– IDATA
– EDATA
– FDATA
– XDATA

5.15 IDATA

Function:

Define indirect data address

Description:

The IDATA directive defines symbolic addresses for the indirectly addressable internal data memory segment (address space). The numeric value of the address must be between 0 and 255 decimal. Addresses greater than 255 are illegal and will be flagged as an error. The symbol will be of type IDATA and cannot be redefined. The value of <expr> must be known on pass 1!

Syntax:

```
<symbol> IDATA <expr>
```

Example:

```
V24BUF IDATA 080H
```

See Also:

- BIT
- CODE
- DATA
- EDATA, FDATA
- XDATA

5.16 EDATA

Function:

Define extended RAM address

Description:

The EDATA directive defines symbolic addresses for the internal extended RAM memory segment (address space). The numeric value of the address must be between 0 and 65536 decimal. The symbol will be of type EDATA and cannot be redefined. The value of <expr> must be known on pass 1!

Syntax:

```
<symbol> EDATA <expr>
```

Example:

```
FIFO EDATA 1000H
```

See Also:

- BIT
- CODE
- DATA, IDATA
- FDATA
- XDATA

5.17 FDATA

Function:

Define flash data Address

Description:

The FDATA directive defines symbolic addresses for the internal flash (non-volatile) data memory segment (address space). The numeric value of the address must be between 0 and 65536 decimal. The symbol will be of type FDATA and cannot be redefined. The value of <expr> must be known on pass 1!

Syntax:

```
<symbol> FDATA <expr>
```

Example:

```
EEVAR1 FDATA 2400H
```

See Also:

- BIT
- CODE
- DATA
- IDATA
- EDATA
- XDATA

5.18 XDATA

Function:

Define external RAM address

Description:

The XDATA directive defines symbolic addresses for the external data memory segment (address space). The numeric value of the address must be between 0 and 65536 decimal. The symbol will be of type XDATA and cannot be redefined. The value of <expr> must be known on pass 1!

Syntax:

```
<symbol> XDATA <expr>
```

Example:

```
SAMPLER XDATA 0100H
```

See Also:

- BIT
- CODE
- DATA
- IDATA
- EDATA, FDATA,

5.19 CSEG

Function:

Switch to CODE segment

Description:

The CSEG directive defines the start of a CODE (program memory) Segment. An assembly source file can consist of several CODE Segments, which are concatenated into one CODE Segment when assembled. The default segment type is CODE. The CODE Segments have their own location counter. If a segment base address is specified with "AT <expr>", a new absolute segment is started, and the location counter is set to <expr>. If "AT <expr>" is omitted, the location counter keeps the previous value of the particular segment. The value of <expr> must be known on pass 1! The ORG directive can also be used to place code and constants at specific locations in the program memory.

At program start, the base address and location counter of the CODE segment are set to zero.

Syntax:

```
 CSEG [AT <expr>]
.CSEG [AT <expr>]
```

Example:

```
CSEG AT 8000h ;start a new CODE segment at address 8000H
CSEG ;switch to previous CODE segment
```

See Also:

- BSEG, DSEG
- ESEG, FSEG
- ISEG, XSEG

5.20 DSEG

Function:

Switch to DATA segment

Description:

The DSEG directive defines the start of a DATA (internal direct data memory) Segment. An assembly source file can consist of several DATA Segments, which are concatenated into one DATA Segment when assembled. The DATA Segments have their own location counter. If a segment base address is specified with "AT <expr>", a new absolute segment is started, and the location counter is set to <expr>. If "AT <expr>" is omitted, the location counter keeps the previous value of the particular segment. The numeric value of <expr> must be between 0 and 255 decimal, although in general <expr> will usually be less than 128. The value of <expr> must be known on pass 1! The ORG directive can also be used to place variables at specific locations in the internal RAM.

At program start, the base address and location counter of the DATA segment are set to zero.

Syntax:

```
 DSEG [AT <expr>]
.DSEG [AT <expr>]
```

Example:

```
DSEG AT 40h ;start a new DATA segment at address 40H
DSEG ;switch to previous DATA segment
```

See Also:

- BSEG, CSEG, ESEG
- FSEG, ISEG, XSEG

5.21 ISEG

Function:

Switch to IDATA segment

Description:

The ISEG directive defines the start of a IDATA (internal direct data memory) Segment. An assembly source file can consist of several IDATA Segments, which are concatenated into one IDATA Segment when assembled. The IDATA Segments have their own location counter. If a segment base address is specified with "AT <expr>", a new absolute segment is started, and the location counter is set to <expr>. If "AT <expr>" is omitted, the location counter keeps the previous value of the particular segment. The numeric value of <expr> must be between 0 and 255 decimal. The value of <expr> must be known on pass 1! The ORG directive can also be used to place variables at specific locations in the internal RAM.

At program start, the base address and location counter of the IDATA segment are set to zero.

Syntax:

```
 ISEG [AT <expr>]
.ISEG [AT <expr>]
```

Example:

```
ISEG AT 80h ;start a new IDATA segment at address 80H
ISEG ;switch to previous IDATA segment
```

See Also:

- BSEG, CSEG, DSEG
- ESEG, FSEG, XSEG

5.22 ESEG

Function:

Switch to EDATA segment

Description:

The ESEG directive defines the start of a EDATA (internal extended data memory) Segment. An assembly source file can consist of several EDATA Segments, which are concatenated into one EDATA Segment when assembled. The EDATA Segments have their own location counter. If a segment base address is specified with "AT <expr>", a new absolute segment is started, and the location counter is set to <expr>. If "AT <expr>" is omitted, the location counter keeps the previous

value of the particular segment. The numeric value of <expr> must be between 0 and 65536 decimal. The value of <expr> must be known on pass 1! The ORG directive can also be used to place variables at specific locations in the internal extended RAM.

At program start, the base address and location counter of the EDATA segment are set to zero.

Syntax:

```
 ESEG [AT <expr>]
.ESEG [AT <expr>]
```

Example:

```
ESEG AT 1000h ;start a new EDATA segment at address 1000H
ESEG ;switch to previous EDATA segment
```

See Also:

- BSEG, CSEG
- DSEG, FSEG
- ISEG, XSEG

5.23 FSEG

Function:

Switch to FDATA segment

Description:

The FSEG directive defines the start of a FDATA (internal non-volatile data memory) Segment. An assembly source file can consist of several FDATA Segments, which are concatenated into one EDATA Segment when assembled. The FDATA Segments have their own location counter. If a segment base address is specified with "AT <expr>", a new absolute segment is started, and the location counter is set to <expr>. If "AT <expr>" is omitted, the location counter keeps the previous value of the particular segment. The numeric value of <expr> must be between 0 and 65536 decimal. The value of <expr> must be known on pass 1! The ORG directive can also be used to place constants and variables at specific locations in the internal non-volatile data memory (Data Flash or EEPROM).

At program start, the base address and location counter of the FDATA segment are set to zero.

Syntax:

```
 FSEG [AT <expr>]
.FSEG [AT <expr>]
```

Example:

```
FSEG AT 1000h ;start a new FDATA segment at address 1000H
FSEG ;switch to previous FDATA segment
```

See Also:

- BSEG, CSEG
- DSEG, ESEG
- ISEG, XSEG

5.24 BSEG

Function:

Switch to BIT segment

Description:

The BSEG directive defines the start of a BIT (internal bit addressable data memory) Segment. An assembly source file can consist of several BIT Segments, which are concatenated into one BIT Segment when assembled. The BIT Segments have their own location counter. If a segment base address is specified with "AT <expr>", a new absolute segment is started, and the location counter is set to <expr>. If "AT <expr>" is omitted, the location counter keeps the previous value of the particular segment. The numeric value of <expr> must be between 0 and 255 decimal, although in general <expr> will usually be less than 128. The value of <expr> must be known on pass 1! The ORG directive can also be used to place bit variables at specific locations in the internal RAM.

At program start, the base address and location counter of the BIT segment are set to zero.

Syntax:

```
 BSEG [AT <expr>]
.BSEG [AT <expr>]
```

Example:

```
BSEG AT 40h ;start a new BIT segment at address 40H
BSEG ;switch to previous BIT segment
```

See Also:

- CSEG, DSEG
- ESEG, FSEG
- ISEG, XSEG

5.25 XSEG

Function:

Switch to XDATA segment

Description:

The XSEG directive defines the start of a XDATA (external data memory) Segment. An assembly source file can consist of several XDATA Segments, which are concatenated into one EDATA Segment when assembled. The XDATA Segments have their own location counter. If a segment base address is specified with "AT <expr>", a new absolute segment is started, and the location counter is set to <expr>. If "AT <expr>" is omitted, the location counter keeps the previous value of the particular segment. The numeric value of <expr> must be between 0 and 65536 decimal. The value of <expr> must be known on pass 1! The ORG directive can also be used to place variables at specific locations in the external RAM.

At program start, the base address and location counter of the XDATA segment are set to zero.

Syntax:

```
 XSEG [AT <expr>]
.XSEG [AT <expr>]
```

Example:

```
XSEG at 0 ;start a new XDATA segment at address 0
XSEG ;switch to previous XDATA segment
```

See Also:

- BSEG, CSEG
- DSEG, ISEG
- ESEG, FSEG

5.26 SET

Function:

Define numeric variable

Description:

The SET instruction defines a symbol for a numeric constant or a register. If a numeric expression <expr> is assigned to the symbol, it will be of the type NUMBER. If a register <reg> is assigned to the symbol, it will be of the type REGISTER. <reg> may be one of the special assembler symbols A, R0, R1, R2, R3, R3, R4, R5, R6, or R7. The SET instruction is similar to EQU; however, symbols defined with SET can be redefined with subsequent SET instructions! The values of <expr> and <reg> must be known on pass 1! A symbol that has been SET, cannot be redefined with EQU! A symbol that has been EQU'd cannot be reSET!

On pass 2, forward references to a SET symbol always evaluate to the last value, the symbol has been SET to on pass 1. Register symbols can be used as instruction operands within the whole program instead of the corresponding registers. Forward references to register symbols are not allowed!

Syntax:

```
<symbol> SET <expr>
<symbol> SET <reg>
.SET <symbol> , <expr>
.SET <symbol> , <reg>
.SET <symbol> = <expr>
.SET <symbol> = <reg>
```

Example:

```
CHAPTER SET 1
CHAPTER SET CHAPTER+1
CHAPTER SET A
.SET POINTER, R0
.SET POINTER = R1
```

See Also:

- DEFINE
- EQU
- #UNDEF

5.27 #UNDEF

Function:

Undefine symbol

Description:

The UNDEF instruction undefines a previously defined symbol. If <symnbol> is not previously defined the directive is silently ignored. This is intended to support configuration management and can be applied sensefully with conditional assembly and include files only.

Syntax:

```
#UNDEF <symbol>
```

Example:

```
#UNDEF POINTER
```

See Also:

- DEFINE
- EQU
- SET

5.28 IF

Function:

Assemble if TRUE

Description:

The IF statement begins an IF-ELSE-ENDIF construct that is used for conditional program assembly. If the expression <expr> is TRUE (not-zero) the IF block is assembled, otherwise any subsequent ELSEIF or ELSE blocks are evaluated. The value of <expr> must be known on pass 1!

IF blocks are terminated by ELSEIF, ELSE, or ENDIF statements.

Syntax:

```
IF <expr>
```

Example:

```
IF A = 1
   . . .
ELSEIF A = 2
   . . .
ELSE
   . . .
ENDIF
```

See Also:

- IFN, ENDIF
- ELSE, ELSEIF

5.29 IFN

Function:

Assemble if FALSE

Description:

The IFN statement begins an IFN-ELSE-ENDIF construct that is used for conditional program assembly. If the expression <expr> is FALSE (zero) the IFN block is assembled, otherwise any subsequent ELSEIF or ELSE blocks are evaluated. IFN is equivalent to IF NOT <expr>. The value of <expr> must be known on pass 1!

IFN blocks are terminated by ELSEIF, ELSE, or ENDIF statements.

Syntax:

```
IFN <expr>
```

Example:

```
IFN NO_TITLE == 1
   $TITLE(My program)
ENDIF
```

See Also:

- IF, ENDIF
- ELSE, ELSEIF

5.30 IFDEF

Function:

Assemble if defined

Description:

The IFDEF statement begins an IFDEF-ELSE-ENDIF construct that is used for conditional program assembly. If <symbol> is defined in the program the IFDEF block is assembled, otherwise any subsequent ELSEIF or ELSE blocks are evaluated. Forward references to <symbol> are not allowed.

IFDEF blocks are terminated by ELSEIFDEF, ELSE, or ENDIF statements.

Syntax:

```
IFDEF <symbol>
```

Example:

```
IFDEF USE_XRAM
   ORL AUXR, #EXRAM ; enable EXRAM bit
ENDIF
```

See Also:

- IFNDEF, ENDIF
- ELSE, ELSEIFDEF

5.31 IFNDEF

Function:

Assemble if not defined

Description:

The IFNDEF statement begins an IFNDEF-ELSE-ENDIF construct that is used for conditional program assembly. If <symbol> is not defined in the program the IFNDEF block is assembled, otherwise any subsequent ELSEIF or ELSE blocks are evaluated. Forward references to <symbol> are not allowed.

IFNDEF blocks are terminated by ELSEIFDEF, ELSE, or ENDIF statements.

Syntax:

```
IFNDEF <symbol>
```

Example:

```
IFNDEF USE_XRAM
   ANL AUXR, #NOT EXRAM ; clear EXRAM bit
ENDIF
```

See Also:

- IFDEF
- ENDIF
- ELSE
- ELSEIFDEF
- ELSEIFNDEF

5.32 IFB

Function:

Assemble if blank

Description:

The IFB statement begins an IFB-ELSE-ENDIF construct that is used for conditional program assembly. If the literal string <literal> is empty the IFB block is assembled, otherwise any subsequent ELSEIF or ELSE blocks are evaluated. <literal> is an unquoted string and must be enclosed in angle brackets (<. . .>).

IFB statements can only be applied sensibly in macro bodies. IFB and IFNB are used to decide whether macro arguments have been left blank, or not.

IFB blocks are terminated by ELSEIFB, ELSE, or ENDIF statements.

Syntax:

```
IFB < <literal> >
```

Example:

```
SUM MACRO A1, B1, C1
IFB <A1>
   IFB <B1>
      IFB <C1>
         $ERROR(At least one operand required for SUM)
      ELSE
         MOV A, C1
      ENDIF
   ELSE
      MOV A, B1
      IFNB <C1>
         ADD A, C1
      ENDIF
   ENDIF
ELSE
   MOV A, A1
   IFNB <B1>
      ADD A, B1
   ENDIF
   IFNB <C1>
      ADD A, C1
   ENDIF
ENDIF
ENDM
```

See Also:

- IFNB, ENDIF
- ELSE, ELSEIFB

5.33 IFNB

Function:

Assemble if not blank

Description:

The IFNB statement begins an IFNB-ELSE-ENDIF construct that is used for conditional program assembly. If the literal string <literal> is empty the IFNB block is assembled, otherwise any subsequent ELSEIF or ELSE blocks are evaluated. <literal> is an unquoted string and must be enclosed in angle brackets (< . . . >).

IFNB statements can only be applied sensibly in macro bodies. IFB and IFNB are used to decide whether macro arguments have been left blank, or not.

IFNB blocks are terminated by ELSEIFB, ELSE, or ENDIF statements.

Syntax:

```
IFNB < <literal> >
```

Example:

```
SUM MACRO A1, B1, C1
IFNB <A1>
   MOV A, A1
   IFNB <B1>
      ADD A, B1
   ENDIF
   IFNB <C1>
      ADD A, C1
   ENDIF
ELSEIFNB <B1>
   MOV A, B1
   IFNB <C1>
      ADD A, C1
   ENDIF
ELSEIFNB <C1>
   MOV A, C1
ELSE
   $ERROR(At least one operand required for SUM)
ENDIF
ENDM
```

See Also:

- IFB, ENDIF
- ELSE, ELSEIFB

5.34 #IF

Function:

Assemble if TRUE

Description:

The #IF statement begins a C-Preprocessor style #IF-#ELSE-#ENDIF construct that is used for conditional program assembly. If the expression <expr> is TRUE (not-zero) the #IF block is assembled, otherwise any subsequent #ELIF or #ELSE blocks are evaluated. The value of <expr> must be known on pass 1!

#IF blocks are terminated by #ELIF, #ELSE, or #ENDIF statements.

Syntax:

`#IF` <expr>

Example:

```
#IF A = 1
   . . .
#ELIF A = 2
   . . .
#ELSE
   . . .
#ENDIF
```

See Also:

– #ELSE, #ELIF, #ENDIF

5.35 #IFDEF

Function:

Assemble if defined

Description:

The #IFDEF statement begins a C-Preprocessor style #IFDEF-#ELSE-#ENDIF construct that is used for conditional program assembly. If <symbol> is defined in the program the #IFDEF block is assembled, otherwise any subsequent #ELIF or #ELSE blocks are evaluated. Forward references to <symbol> are not allowed.

#IFDEF blocks are terminated by #ELIF, #ELSE, or #ENDIF statements.

Syntax:

#IFDEF <symbol>

Example:

```
#IFDEF USE_XRAM
   ORL AUXR, #EXRAM ; enable EXRAM bit
#ENDIF
```

See Also:

– #IFNDEF, #ENDIF

– #ELSE, #ELIF

5.36 #IFNDEF

Function:

Assemble if not defined

Description:

The #IFNDEF statement begins a C-Preprocessor style #IFNDEF-#ELSE-#ENDIF construct that is used for conditional program assembly. If <symbol> is not defined in the program the #IFNDEF block is assembled, otherwise any subsequent #ELIF or #ELSE blocks are evaluated. Forward references to <symbol> are not allowed.

#IFNDEF blocks are terminated by #ELIF, #ELSE, or #ENDIF statements.

Syntax:

#IFNDEF <symbol>

Example:

```
#IFNDEF USE_XRAM
   ANL AUXR, #NOT EXRAM ; clear EXRAM bit
#ENDIF
```

See Also:

- #IFDEF, #ENDIF
- #ELSE, #ELIF

5.37 .IF

Function:

Assemble if TRUE

Description:

The .IF statement begins an .IF-.ELSE-.ENDIF construct that is used for conditional program assembly. If the expression <expr> is TRUE (not-zero) the .IF block is assembled, otherwise the subsequent .ELSE block, if present, is evaluated. The value of <expr> must be known on pass 1!

.IF blocks are terminated by .ELSE, or .ENDIF statements.

Syntax:

.IF <expr>

Example:

```
.IF A = 1
. . .
.ELSE
.IF A = 2
. . .
.ELSE
. . .
.ENDIF
.ENDIF
```

See Also:

- #ELSE, #ELIF, #ENDIF

5.38 ELSEIF

Function:

Alternatively assembler if TRUE

Description:

The ELSEIF statement begins an alternate block after an IF or ELSEIF block. If the expression <expr> is TRUE (not-zero) the ELSEIF block is assembled, otherwise any subsequent ELSEIF or ELSE blocks are evaluated. The value of <expr> must be known on pass 1!

ELSEIF blocks are terminated by ELSEIF, ELSE, or ENDIF statements.

Syntax:

ELSEIF <expr>

Example:

```
IF A = 1
   . . .
ELSEIF A = 2
   . . .
ELSE
   . . .
ENDIF
```

See Also:

- IF
- ENDIF
- ELSE
- ELSEIFN

5.39 ELSEIFN

Function:

Alternatively assemble if FALSE

Description:

The ELSEIFN statement begins an alternate block after an IF or ELSEIF block. If the expression <expr> is FALSE (zero) the ELSEIFN block is assembled, otherwise any subsequent ELSEIF or ELSE blocks are evaluated. ELSEIFN is equivalent to ELSEIF NOT <expr>. The value of <expr> must be known on pass 1!

ELSEIFN blocks are terminated by ELSEIF, ELSE, or ENDIF statements.

Syntax:

ELSEIFN <expr>

Example:

```
IF A = 0
   . . .
ELSEIFN A > MAX
   . . .
ELSE
   . . .
ENDIF
```

See Also:

- IFN
- ENDIF
- ELSE
- ELSEIF

5.40 ELSEIFDEF

Function:

Alternatively assemble if defined

Description:

The ELSEIFDEF statement begins an alternate block after an IF or ELSEIF block. If <symbol> is defined in the program the ELSEIFDEF block is assembled, otherwise any subsequent ELSEIF or ELSE blocks are evaluated. Forward references to <symbol> are not allowed.

ELSEIFDEF blocks are terminated by ELSEIF, ELSE, or ENDIF statements.

Syntax:

```
ELSEIFDEF <symbol>
```

Example:

```
IFDEF USE_XRAM
   ORL AUXR, #EXRAM ; enable EXRAM bit
ENDIF
```

See Also:

ELSE, ELSEIFNDEF, ENDIF, IFDEF

5.41 ELSEIFNDEF

Function:

Alternatively assemble in not defined

Description:

The ELSEIFNDEF statement begins an alternate block after an IF or ELSEIF block. If <symbol> is not defined in the program the ELSEIFNDEF block is assembled, otherwise any subsequent ELSEIF or ELSE blocks are evaluated. Forward references to <symbol> are not allowed.

ELSEIFNDEF blocks are terminated by ELSEIF, ELSE, or ENDIF statements.

Syntax:

```
ELSEIFNDEF <symbol>
```

Example:

```
IFNDEF USE_XRAM
   ANL AUXR, #NOT EXRAM ; clear EXRAM bit
ENDIF
```

See Also:

ELSE, ELSEIFDEF, ENDIF, IFNDEF

5.42 ELSEIFB

Function:

Alternatively assemble if blank

Description:

The IFB statement begins an IFB-ELSE-ENDIF construct that is used for conditional program assembly. If the literal string <literal> is empty the IFB block is assembled, otherwise any subsequent ELSEIF or ELSE blocks are evaluated. <literal> is an unquoted string and must be enclosed in angle brackets.

IFB statements can only be applied sensibly in macro bodies. IFB and IFNB are used to decide whether macro arguments have been left blank, or not.

ELSEIFB blocks are terminated by ELSEIF, ELSE, or ENDIF statements.

Syntax:

```
ELSEIFB < <literal> >
```

Example:

```
SUM MACRO A1, B1, C1
IFB
IFB
IFB
$ERROR(At least one operand required for SUM)
ELSE
MOV A, C1
ENDIF
ELSE
MOV A, B1
IFNB
ADD A, C1
ENDIF
ENDIF
ELSE
MOV A, A1
IFNB
ADD A, B1
ENDIF
IFNB
ADD A, C1
ENDIF
ENDIF
ENDM
```

See Also:

ELSE, ELSEIFB, ENDIF, IFNB

5.43 ELSEIFNB

Function:

Alternatively assemble if not blank

Description:

The IFNB statement begins an IFNB-ELSE-ENDIF construct that is used for conditional program assembly. If the literal string <literal> is empty the IFNB block is assembled, otherwise any subsequent ELSEIF or ELSE blocks are evaluated. <literal> is an unquoted string and must be enclosed in angle brackets.

IFNB statements can only be applied sensibly in macro bodies. IFB and IFNB are used to decide whether macro arguments have been left blank, or not.

IFNB blocks are terminated by ELSEIF, ELSE, or ENDIF statements.

Syntax:

```
ELSEIFNB < <literal> >
```

Example:

```
SUM MACRO A1, B1, C1
IFNB
MOV A, A1
IFNB
ADD A, B1
ENDIF
IFNB
ADD A, C1
ENDIF
ELSEIFNB
MOV A, B1
IFNB
ADD A, C1
ENDIF
ELSEIFNB
MOV A, C1
ELSE
$ERROR(At least one operand required for SUM)
ENDIF
ENDM
```

See Also:

ELSE, ELSEIFB, ENDIF, IFNB

5.44 #ELIF

Function:

Alternatively assemble if TRUE

Description:

The #IF statement begins a C-Preprocessor style #IF-#ELSE-#ENDIF construct that is used for conditional program assembly. If the expression <expr> is TRUE (not-zero) the #IF block is assembled, otherwise any subsequent #ELIF or #ELSE blocks are evaluated. The value of <expr> must be known on pass 1!

#ELIF blocks are terminated by #ELIF, #ELSE, or #ENDIF statements.

Syntax:

#ELIF <expr>

Example:

```
#IF A = 1
. . .
#ELIF A = 2
. . .
#ELSE
. . .
#ENDIF
```

See Also:

#ELSE, #ELIF, #ENDIF

5.45 ELSE

Function:

Alternatively assemble block

Description:

The ELSE statement introduces an alternate program block after an IF or ELSEIF statement. The ELSE block is assembled only if the prior IF or ELSEIF condition expression was false.

ELSE blocks are terminated by ENDIF statements.

Syntax:

ELSE

Example:

```
IF A = 1
. . .
ELSEIF A = 2
. . .
ELSE
. . .
ENDIF
```

See Also:

IF, ELSEIF, ENDIF

5.46 #ELSE

Function:

Alternatively assemble block

Description:

The #ELSE statement introduces an alternate program block after an #IF or #ELIF statement. The #ELSE block is assembled only if the prior #IF or #ELIF condition expression was false.

#ELSE blocks are terminated by #ENDIF statements.

Syntax:

#ELSE

Example:

```
#IF A = 1
. . .
#ELSEIF A = 2
. . .
#ELSE
. . .
#ENDIF
```

See Also:

#IF, #ELIF, #ENDIF

5.47 .ELSE

Function:

Alternatively assemble block

Description:

The .ELSE statement introduces an alternate program block after an .IF statement. The .ELSE block is assembled only if the prior .IF condition expression was false.

.ELSE blocks are terminated by .ENDIF statements.

Syntax:

.ELSE

Example:

```
.IF A = 1
. . .
.ELSE
. . .
.ENDIF
```

See Also:

.IF, .ENDIF

5.48 ENDIF

Function:

Terminate an IF conditional block

Description:

The ENDIF statement terminates an IF-ELSE-ENDIF statement construct. When the assembler encounters an ENDIF statement, it concludes processing the IF block and resumes assembly on the next line which may involve continuing in another IF block if the current block was nested.

The ENDIF statement must be preceded by an IF, ELSEIF, or ELSE block.

Syntax:

ENDIF

Example:

```
IF A = 1
. . .
ELSEIF A = 2
. . .
ELSE
. . .
ENDIF
```

See Also:

IF, ELSEIF, ELSE

5.49 #ENDIF

Function:

Terminate a #IF conditional block

Description:

The #ENDIF statement terminates an #IF-#ELSE-#ENDIF statement construct. When the assembler encounters an #ENDIF statement, it concludes processing the #IF block and resumes assembly on the next line which may involve continuing in another IF block if the current block was nested.

The #ENDIF statement must be preceded by an #IF, #ELIF, or #ELSE block.

Syntax:

#ENDIF

Example:

```
#IF A = 1
. . .
#ELIF A = 2
. . .
#ELSE
. . .
#ENDIF
```

See Also:

#IF, #ELIF, #ELSE

5.50 .ENDIF

Function:

Terminate a .IF conditional block

Description:

The .ENDIF statement terminates an .IF-.ELSE-.ENDIF statement construct. When the assembler encounters an .ENDIF statement, it concludes processing the .IF block and resumes assembly on the next line which may involve continuing in another IF block if the current block was nested.

The .ENDIF statement must be preceded by an .IF or .ELSE block.

Syntax:

```
.ENDIF
```

Example:

```
.IF A = 1
. . .
.ELSE
. . .
.ENDIF
```

See Also:

.IF, .ELSE

5.51 MACRO

Function:

Define a macro

Description:

Defines a callable macro called <macro name> with optional parameters separated by commas. All parameter names of a macro must be valid symbols and unique within the macro (parameter names are local to the macro only). Keywords cannot be used as parameter names. A call to <macro name> will expand to <macro body>.

Macro definitions are terminated by ENDM statements.

Syntax:

```
<macro name> MACRO [<parameter 1>[, <parameter 2>[, ... ,<parameter n>]]]
. . .
<macro body>
. . .
ENDM
```

Example:

```
SERIES MACRO STARTV, ENDV
IF STARTV LE ENDV
DB STARTV
SERIES %STARTV+1, ENDV
ENDIF
ENDM

$GEN CONDONLY
SERIES 1,3
DB 1
```

```
DB 2
DB 3
```

See Also:

ENDM, EXITM, LOCAL, REPT

5.52 LOCAL

Function:

Declare a local symbol

Description:

Defines one or more locally scoped labels, separated by commas. Local labels must be valid symbols, unique within the macro, and different from the formal parameters (if any). Keywords cannot be used as local symbol names. If a local symbol has the same name as a global symbol, the local scope takes precedance during substitution.

LOCAL statements are only valid at the start of a macro defintion, before any body lines are defined.

Syntax:

LOCAL <symbol 1>[, <symbol 2>[, ... ,<symbol n>]]

Example:

```
CJLT MACRO OP1, OP2, TARGET
LOCAL NEQ, NO_JMP
CJNE OP1, OP2, NEQ
SJMP NO_JMP
NEQ: JC TARGET
NO_JMP:
ENDM
```

See Also:

ENDM, EXITM, MACRO, REPT

5.53 ENDM

Function:

Terminate a macro or repeat definition

Description:

Terminates a MACRO or REPT macro defintion block. ENDM statements must be preceded by a MACRO or REPT statement.

Syntax:

ENDMACRO
ENDM

Example:

```
SIMPLE MACRO
NOP
NOP
ENDM
```

See Also:

EXITM, LOCAL, MACRO, REPT

5.54 EXITM

Function:

Exit macro expansion

Description:

The EXITM directive immediately terminates expansion of the current macro. When EXITM is detected, the macro processor stops expanding the current macro prematurely and resumes processing after the next ENDM directive. The EXITM directive is only useful in conditional statements.

Syntax:

```
EXITM
```

Example:

```
MYADD MACRO OP1, OP2
MOV A, OP1
IFB
EXITM
ELSE
ADD A, OP2
ENDIF
ENDM
```

See Also:

ENDM, LOCAL, MACRO, REPT

5.55 REPT

Function:

Define a repeat macro

Description:

Defines a repeat macro. The <macro body> will be expanded <expr> times immediately after the ENDM statement. The value of <expr> must be known on pass 1!

Repeat definitions are terminated by ENDM statements.

Syntax:

```
REPT <expr>
. . .
<macro body>
```

```
. . .
ENDM
```

Example:

```
REPT 4
NOP
ENDM
NOP
NOP
NOP
NOP
```

See Also:

ENDM, EXITM, LOCAL, MACRO

Section 6

Assembler Controls

The following table lists all the implemented controls and their abbreviations. In the subsequent paragraphs, all C51ASM assembler directives are described. Lexical symbols are written in lower case letters, while assembler keywords are written in upper case. Instruction arguments are represented by <arg>, <arg1> or something like that. Numeric expressions are represented by <expr>, <expr1> and so on. Syntax elements enclosed in brackets are optional. The ellipsis "..." means always "a list with any number of elements". Primary controls (P) can only be used at the beginning of the program and remain in effect throughout the assembly. General controls (G) may be used everywhere in the program. See Section 3.9 ”Assembler Controls” on page 3-12.

Table 6-1. C51ASM Controls

Control	Type	Abbreviation	Description
$COND	G		list full IF .. ENDIF constructions
$CONDONLY	G		list assembled lines only
$DATE	P	$DA	inserts date string into page header
$DEBUG	P	$DB	include debug information into object
$DEVICE	P		select device
$EJECT	G	$EJ	start a new page in list file
$ERROR	G		force a user-defined error
$GEN	G	$GE	list macro calls and expansion lines
$GENONLY	G	$GO	list expansion lines only
$INCDIR	P		add a directory to search path
$INCLUDE	G	$IC	include a source file
$LIST	G	$LI	list subsequent source lines
$MACRO	P	$MR	reserve n % of free memory for macros
$MAPFSEG	P		remap FSEG location in object file
$MESSAGE	G		output a message to console
$MOD51	P	$MO	enable predefined 80C51 SFR symbols
$MOD52	P		enable predefined 80C52 SFR symbols
$NOBUILTIN	P		don't list predefined symbols
$NOCOND	G		don't list lines in false branches
$NODEBUG	P	$NODB	don't include debug information
$NOGEN	G	$NOGE	list macro calls only
$NOLIST	G	$NOLI	don't list subsequent source lines
$NOMACRO	P	$NOMR	reserve all for the symbol table

Table 6-1. C51ASM Controls

Control	Type	Abbreviation	Description
$NOMOD51	P	$NOMO	disable predefined SFR symbols
$NOOBJECT	P	$NOOJ	don't generate object file
$NOPAGING	P	$NOPI	disable listing page formatting
$NOPRINT	P	$NOPR	don't generate list file
$NOSYMBOLS	P	$NOSB	don't create symbol table
$NOTABS	P		don't use tabs in list file
$NOXREF	P	$NOXR	don't create cross reference
$OBJECT	P	$OJ	generate object file
$PAGELENGTH	P	$PL	set lines per page for listing
$PAGEWIDTH	P	$PW	set columns per line for listing
$PAGING	P	$PI	enable listing page formatting
$PRINT	P	$PR	generate list file
$RESTORE	G	$RS	restore old $LIST/$GEN/$COND state
$SAVE	G	$SA	save current $LIST/$GEN/$COND state
$SYMBOLS	P	$SB	create symbol table
$TITLE	G	$TT	inserts title string into page header
$WARNING	G		output a warning message to console
$XREF	P	$XR	create cross reference

6.1 $DATE

Function:

Insert Date in Header

Description:

Inserts a date string into the list file page header. If $DATE() or $DATE is specified, the actual date is inserted. Date strings will be truncated to a maximum length of 11 characters. Default is: no date string. The control has no effect, when the $NOPAGING control has been specified.

Syntax:

```
$DATE [(string)]
```

Abbreviation:

```
$DA
```

Example:

```
$DATE(10/09/2008) ;date of latest version
$DATE ;use today's date
$DA(1-JUL-1984)
```

See Also:

– $PAGING, $TITLE

6.2 $DEBUG

Function:

Generate Debugging Information

Description:

Includes debug information into the OMF-51 module. $DEBUG has no effect unless -g/--debug or --omf-51 was specified on the command line. The string if specified will be used for the OMF-51 file name unless previously set by the --omf-51 option.

Syntax:

$DEBUG [(string)]

Abbreviation:

```
$DB
```

Example:

```
$DEBUG ;place debug information in <srcfile>.omf
$DB(my_asm.omf) ;place debug information in my_asm.omf
```

See Also:

– $NODEBUG

6.3 $NODEBUG

Function:

Suppress Debug Info

Description:

Don't include debug information. (Default!) $NODEBUG is overridden by the --debug option.

Syntax:

$NODEBUG

Abbreviation:

```
$NODB
```

Example:

```
$NODEBUG ;do not generate debug file
$NODB
```

See Also:

– $DEBUG

6.4 $DEVICE

Function:

Select Device

Description:

Assemble with the specified device model. $DEVICE is overridden by the --target option.

Syntax:

```
$DEVICE (string)
```

Abbreviation:

None

Example:

```
$DEVICE(AT89LP828) ;select at89lp828 model
```

See Also:

None

6.5 $INCDIR

Function:

Include Directory

Description:

The specified string is added to the include directory search path after any directories specified on the command line with the --include option.

Syntax:

```
$INCDIR (string)
```

Abbreviation:

None

Example:

```
$INCDIR(C:\ASM\INC) ; look for includes in C:\ASM\INC
```

See Also:

– $INCLUDE

6.6 $MACRO

Function:

Expand Macros

Description:

Save macro definitions and expand macro calls. (Default!) Optionally reserve n % of free memory for macro definitions. (0 <= n <= 100) Default is n=50. The control has been implemented for compatibility purposes only. In C51ASM it has no effect except that it cancels the $NOMACRO control.

Syntax:

$MACRO [(n)]

Abbreviation:

$MR

Example:

```
$MACRO
$MR(10)
```

See Also:

– $NOMACRO

6.7 $NOMACRO

Function:

Suppress Macro Expansion

Description:

Don't save macro definitions and don't expand macro calls. Reserve all free memory for the symbol table. The control has been implemented for compatibility purposes only. In C51ASM, it only suppresses the macro expansion.

Syntax:

$NOMACRO

Abbreviation:

$NOMR

Example:

```
$NOMACRO
$NOMR
```

See Also:

– $MACRO

6.8 $MAPFSEG

Function:

Remap FSEG location in Object File

Description:

Many device programmers map nonvolatile data segments to some location in their input buffers. Use $MAPFSEG to move FSEG data to a different location in the object file. <offset> specifies the address in the object file where the FSEG data will start. <start> optionally specifies the initial address in FSEG that will map to <offset>. If a device is explicitly specified with $DEVICE or --target, <start> should be limited to either 0 or the normal starting address of FDATA. <offset> and <start> both default to 0.

Syntax:

```
$MAPFSEG (offset[:start])
```

Abbreviation:

None

Example:

```
$MAPFSEG(1000H) ; output FSEG at 1000H starting from 0
$MAPFSEG(2000H:200H) ; output FSEG at 2000H starting from 200H
```

See Also:

None

6.9 $MOD51

Function:

Use 8051 Predefined Symbols

Description:

Switches on the built-in 8051 special function register and interrupt symbol definitions. (Default!)

Syntax:

```
$MOD51
```

Abbreviation:

```
$MO
```

Example:

```
$MOD51
$MO
```

See Also:

- $MOD52
- $NOMOD51

6.10 $MOD52

Function:

Use 8052 Predefined Symbols

Description:

Switches on the built-in 8052 special function register and interrupt symbol definitions.

Syntax:

$MOD52

Abbreviation:

None

Example:

```
$MOD51
$MO
```

See Also:

- $MOD51
- $NOMOD51

6.11 $NOMOD51

Function:

Suppress Predefined Symbols

Description:

Switches off the built-in 8051 special function register and interrupt symbol definitions. The predefined symbols ??C51ASM and ??VERSION cannot be switched off!

Syntax:

$NOMOD51

Abbreviation:

```
$NOMO
```

Example:

```
$NOMOD51 ;switch off 8051 SFR symbol definitions
$NOMO
```

See Also:

- $MOD51
- $MOD52

6.12 $OBJECT

Function:

Generate Object File

Description:

Generate an object file. (Default!) By default the output format is Intel HEX unless changed on the command line with --filetype. The string if specified is used for the object file name unless previously set by the --outfile option. The $OBJECT control is overridden by the --noobject option.

Syntax:

$OBJECT [(string)]

Abbreviation:

$OJ

Example:

```
$OBJECT
$OJ(my_asm.hex)
```

See Also:

– $NOOBJECT

6.13 $NOOBJECT

Function:

Suppress Object File

Description:

Don't generate an object file. The $NOOBJECT control is overridden by the --outfile option.

Syntax:

$NOOBJECT

Abbreviation:

$NOOJ

Example:

```
$NOOBJECT
$NOOJ
```

See Also:

– $OBJECT

6.14 $PAGING

Function:

Page Format List File

Description:

Switches on the page formatting in the list file. (Default!) The $PAGING control is overridden by the --nopaging option.

Syntax:

`$PAGING`

Abbreviation:

`$PI`

Example:

```
$PAGING
$PI
```

See Also:

- $NOPAGING
- $PAGELENGTH
- $PAGEWIDTH

6.15 $NOPAGING

Function:

Suppress Page Formatting

Description:

Switches off the page formatting in the list file.

Syntax:

`$NOPAGING`

Abbreviation:

`$NOPI`

Example:

```
$NOPAGING
$NOPI
```

See Also:

- $PAGING

6.16 $PAGELENGTH

Function:

Set Page Length

Description:

Sets the list file page length to n lines. (12 <= n <= 65535) Default is n=64. The control has no effect, when the $NOPAGING control has been specified.

Syntax:

```
$PAGELENGTH (n)
```

Abbreviation:

```
$PL
```

Example:

```
$PAGELENGTH(60) ;set page length to 60 lines per page
$PL(72) ;set page length to 72 lines per page
```

See Also:

- $EJECT
- $NOPAGING
- $PAGING
- $PAGEWIDTH

6.17 $PAGEWIDTH

Function:

Set Page Width

Description:

Sets the list file page width to n columns. (72 <= n <= 255) Default is n=132.

Syntax:

```
$PAGEWIDTH (n)
```

Abbreviation:

```
$PW
```

Example:

```
$PAGEWIDTH(72) ;set page width to 72 characters per line
$PW(80) ;set page width to 80 characters per line
```

See Also:

- $NOPAGING
- $PAGELENGTH
- $PAGING

6.18 $PRINT

Function:

Generate Listing File

Description:

Generate a listing file. (Default!) The string if specified is used for the listing file name, unless previously set by the --listfile option. The $PRINT control is overridden by the --nolist option.

Syntax:

`$PRINT` `[(string)]`

Abbreviation:

`$PR`

Example:

```
$PRINT
$PR(my_asm.lst)
```

See Also:

– $NOPRINT

6.19 $NOPRINT

Function:

Suppress Listing File

Description:

Don't generate a listing file. The $NOPRINT control is overridden by the --listfile option.

Syntax:

`$NOPRINT`

Abbreviation:

`$NOPR`

Example:

```
$NOPRINT
$NOPR
```

See Also:

– $PRINT

6.20 $SYMBOLS

Function:

Generate Symbol Table

Description:

Generates the symbol table at the end of the list file. (Default!) When the $XREF control is active, $SYMBOLS has no effect!

Syntax:

`$SYMBOLS`

Abbreviation:

`$SB`

Example:

```
$SYMBOLS
$SB
```

See Also:

- $NOBUILTIN
- $NOSYMBOLS
- $XREF

6.21 $NOSYMBOLS

Function:

Suppress Symbol Table

Description:

Suppresses the symbol table at the end of the list file. When the $XREF control is active, $NOSYMBOLS has no effect!

Syntax:

`$NOSYMBOLS`

Abbreviation:

`$NOSB`

Example:

```
$NOSYMBOLS ;no symbol table required
$NOSB
```

See Also:

- $SYMBOLS
- $XREF

6.22 $NOBUILTIN

Function:

Hide Predefined Symbols

Description:

Suppresses the predefined (built-in) symbols in the symbol table or cross-reference listing for a better survey. Only the user-defined symbols are listed.

Syntax:

$NOBUILTIN

Abbreviation:

None

Example:

```
$NOBUILTIN ; hide predefined symbols
```

See Also:

- $SYMBOLS
- $XREF

6.23 $NOTABS

Function:

Expand Tabs

Description:

Expands all tab characters in the list file output to blanks.

Syntax:

$NOTABS

Abbreviation:

None

Example:

```
$NOTABS ;printer doesn't support tab characters
```

See Also:

None

6.24 $XREF

Function:

Generate Cross-Reference

Description:

Generates a cross-reference listing instead of a symbol table. $XREF overrides $SYMBOLS.

Syntax:

$XREF

Abbreviation:

$XR

Example:

```
$XREF ;generate a cross-reference listing
$XR
```

See Also:

- $NOXREF
- $SYMBOLS

6.25 $NOXREF

Function:

Suppress Cross-Reference

Description:

Generates a symbol table instead of a cross-reference listing (Default!) unless $NOSYMBOLS is also specified.

Syntax:

$NOXREF

Abbreviation:

$NOXR

Example:

```
$NOXREF ;suppress cross-reference table
$NOXR
```

See Also:

- $NOSYMBOLS
- $SYMBOLS
- $XREF

6.26 $COND

Function:

Fully List Conditional Branches

Description:

List full IF .. ELSEIF .. ELSE .. ENDIF constructions. (Default!) The Control is overridden by $NOLIST.

Syntax:

$COND

Abbreviation:

None

Example:

```
$COND
```

See Also:

- $CONDONLY
- $NOCOND
- $NOLIST

6.27 $NOCOND

Function:

Suppress False Conditional Branches

Description:

Don't list lines in false IF .. ELSEIF .. ELSE .. ENDIF branches. The Control is overridden by $NOLIST.

Syntax:

$NOCOND

Abbreviation:

None

Example:

```
$NOCOND
```

See Also:

- $COND
- $CONDONLY
- $NOLIST

6.28 $CONDONLY

Function:

Suppress False Conditional Branches and Conditional Directives

Description:

List lines in true IF .. ELSEIF .. ELSE .. ENDIF branches only, without the IF, ELSEIF, ELSE and ENDIF statements itself. The Control is overridden by $NOLIST.

Syntax:

$CONDONLY

Abbreviation:

None

Example:

```
$CONDONLY
```

See Also:

- $COND
- $NOCOND
- $NOLIST

6.29 $EJECT

Function:

Insert Page Break

Description:

Starts a new page in the list file by inserting a form feed (ASCII 0x0C). The control has no effect, when the $NOPAGING control has been specified.

Syntax:

$EJECT

Abbreviation:

```
$EJ
```

Example:

```
$EJECT ;new page with new title
$EJ
```

See Also:

- $PAGELENGTH
- $PAGING
- $NOPAGING

6.30 $ERROR

Function:

Print User Error Message

Description:

Forces an assembly error with a user-defined error message, and increment the error count. This is intended to support configuration management and can be applied sensefully with conditional assembly only.

Syntax:

```
$ERROR (string)
```

Abbreviation:

None

Example:

```
$ERROR(invalid configuration: buffer size > external RAM size)
```

See Also:

- $MESSAGE
- $WARNING

6.31 $WARNING

Function:

Print User Warning Message

Description:

Outputs a user-defined warning message to the console, and increments the warning count. This is also intended to ease configuration management.

Syntax:

```
$WARNING (string)
```

Abbreviation:

None

Example:

```
$WARNING(int. RAM doesn't meet minimum stack size requirements)
```

See Also:

- $ERROR
- $MESSAGE

6.32 $MESSAGE

Function:

Print User Informational Message

Description:

Outputs a user-defined informational message to the console, but does not increment the warning or error counts.

Syntax:

```
$MESSAGE (string)
```

Abbreviation:

None

Example:

```
$MESSAGE(Using dual datapointers)
```

See Also:

- $ERROR
- $WARNING

6.33 $GEN

Function:

Fully List Macros

Description:

List macro calls and expanded macros. (Default!) The listing fully shows the nesting of macro calls. The Control is overridden by $NOLIST.

Syntax:

```
$GEN
```

Abbreviation:

```
$GE
```

Example:

```
$GEN
$GE
```

See Also:

- $GENONLY
- $NOGEN
- $NOLIST

6.34 $NOGEN

Function:

Suppress Macro Bodies

Description:

List macro calls only. The expanded macros are not listed. The Control is overridden by $NOLIST.

Syntax:

```
$NOGEN
```

Abbreviation:

```
$NOGE
```

Example:

```
$NOGEN
$NOGE
```

See Also:

- $GEN
- $GENONLY

6.35 $GENONLY

Function:

Suppress Macro Calls

Description:

List the expanded macro bodies only. Macro calls and EXITM statements are not listed. The Control is overridden by $NOLIST.

Syntax:

```
$GENONLY
```

Abbreviation:

```
$GO
```

Example:

```
$GENONLY
$GO
```

See Also:

- $GEN
- $NOGEN
- $NOLIST

6.36 $INCLUDE

Function:

Include a File

Description:

Includes an external source file into the assembler program just behind the $INCLUDE statement. If the include file has not been specified with an absolute path, and it cannot be found in the default directory, the path specified with the /INCLUDES (Linux: -i) command line option (if present) is searched from left to right, and if it cannot be found there either, the path specified with the environment variable C51ASMINC (if defined) is searched from left to right as well. Include files may be nested to any depth.

Syntax:

```
$INCLUDE (file)
.INCLUDE (file)
#INCLUDE <file>
#INCLUDE "file"
```

Abbreviation:

```
$IC
```

Example:

```
$INCLUDE(8052.inc) ;include 8052.inc from the search path
$IC(C:\ASM\INC\my_lib.h) ;include my_lib.h from C:\ASM\INC
```

See Also:

– $INCDIR

6.37 $LIST

Function:

List Source Lines

Description:

List source code lines. (Default!)

Syntax:

```
$LIST
```

Abbreviation:

```
$LI
```

Example:

```
$LIST ;switch on listing
$LI
```

See Also:

– $NOLIST

6.38 $NOLIST

Function:

Suppress Source Lines

Description:

Do not list source code lines, provided they do not contain errors, until the next $LIST statement occurs. $NOLIST overrides any $COND, $CONDONLY, $GEN, or $GENONLY controls until listing is enabled again.

Syntax:

$NOLIST

Abbreviation:

```
$NOLI
```

Example:

```
$NOLIST ;switch off listing
$NOLI
```

See Also:

– $LIST

6.39 $SAVE

Function:

Save Listing State

Description:

Saves the current $LIST/$GEN/$COND state on a $SAVE-stack. $SAVE statements can be nested to any depth.

Syntax:

$SAVE

Abbreviation:

```
$SA
```

Example:

```
$SAVE ;save old $LIST/$GEN/$COND status
$SA
```

See Also:

– $RESTORE
– $COND
– $GEN
– $LIST

6.40 $RESTORE

Function:

Restore Listing State

Description:

Restores a previously saved $LIST/$GEN/$COND state.

Syntax:

$RESTORE

Abbreviation:

$RS

Example:

```
$RESTORE ;restore previous listing mode
$RS
```

See Also:

- $SAVE
- $COND
- $GEN
- $LIST

6.41 $TITLE

Function:

Set Page Title

Description:

Inserts a title string into the list file page header. Titles may be truncated according to the specified (or default) page width. Default: C51ASM copyright information. The control has no effect, when the $NOPAGING control has been specified.

Syntax:

$TITLE (string)

Abbreviation:

$TT

Example:

```
$TITLE(My Assembly Program)
$TT(Sample Program)
```

See Also:

- $DATE
- $PAGING

Section 7

Assembler Compatibility

The Atmel® C51ASM assembler implements a subset of the Intel® ASM51 assembly language, plus some extensions. The Intel ASM51 syntax is supported by most 8051 assemblers, therefore a high level of compatibility is possible with existing 8051 assembly source files. All MCS®51 instruction mnemonics are supported. The following sections describe the differences that may exist between C51ASM and other assemblers.

7.1 7.1 Restrictions

C51ASM only generates absolute object files instead of relocatable object modules. The entire object is built in one step without calling a linker. Therefore the entire source code of an application program must reside in a single file and its subsequent include files. The assembler directives that deal with relocatable segments or external or public symbols have not been implemented, but are still reserved. These include:

- PUBLIC
- EXTRN
- SEGMENT
- RSEG

C51ASM does not support the Intel Macro Processing Language (MPL). Only standard callable macros and repeat macros are supported.

7.2 7.2 Extensions

C51ASM includes some features that are not found in other assemblers.

- C51ASM includes support for the extended instruction set of the Atmel AT89LP family and special data pointer mode aliases. See Section 3.7 ”The 8051 Instruction Set” on page 3-9.
- C51ASM adds the additional segment types EDATA (for internal Extended RAM) and FDATA (for internal Flash/EEPROM data memory). This includes the symbol and segment definition directives: EDATA, FDATA, ESEG, and FSEG. An additional object file is created for initialized FDATA values, with the $MAPFSEG control affecting the addressing in the FDATA object file.
- C51ASM includes preliminary support for devices with CODE memories above 64KB. This feature will be fully functional in a future release.

Section 8

Supported Devices

The Atmel® C51ASM assembler can support any 8051 device that uses the standard 8051 instruction set. However, C51ASM was intended, generally, to support the Atmel 8051 family, and specifically, the Atmel AT89LP single-cycle microcontrollers. C51ASM supports devices in two ways: predefined SFR symbols and device models.

C51ASM includes predefined SFR address and bit symbols for the standard base registers of the 80C51 and 80C52 microcontrollers as listed in "Appendix C - Predefined Symbols" . These definitions are common to "most" 8051 devices. Use the $MOD51 and $MOD52 controls to define the symbols for 80C51 and 80C52, respectively. $MOD51 is on by default. To disabled the predefined values and use your own, specify the $NOMOD51control.

C51ASM also includes device model support for Atmel 8051 devices. Device models will set limits on memory segment sizes and features that correspond to the actual physical device. When a target device is selected, the assembler checks the model and will generate errors if a certain feature is not supported, or if memory is allocated out of bounds. Device models are specified by the --target option or $DEVICE control. When no device is specified, a generic model is used. Note that selecting a device model does not predefine any SFR symbols. Device specific header files can be found under the include directory.

Call C51ASM on the command line with the --devices option to print a list of the supported devices. The following device models are supported by C51ASM:

Table 8-1. Supported Devices

	Flash/EEPROM				
Device	**Code**	**Data**	**RAM**	**ERAM**	**XRAM**
(default)	65536	65536	256	65536	Y
AT89LP2052	2048	0	256	0	N
AT89LP4052	4096	0	256	0	N
AT89LP213	2048	0	128	0	N
AT89LP214	2048	0	128	0	N
AT89LP216	2048	0	128	0	N
AT89LP428	4096	512	256	512	N
AT89LP52	8192	256	256	0	Y
AT89LP828	8192	1024	256	512	N
AT89LP6440	65536	8192	256	4096	Y
AT89S2051	2048	0	256	0	N
AT89S4051	4096	0	256	0	N
AT89S51	4096	0	128	0	Y

Table 8-1. Supported Devices

Device	Flash/EEPROM		RAM	ERAM	XRAM
	Code	Data			
AT89S52	8192	0	256	0	Y
AT89S53	12288	0	256	0	Y
AT89S8252	8192	2048	256	0	Y
AT89S8253	12288	2048	256	0	Y
AT89LS51	4096	0	128	0	Y
AT89LS52	8192	0	256	0	Y
AT89C2051	2048	0	128	0	N
AT89C4051	4096	0	128	0	N
AT89C51	4096	0	128	0	Y
AT89C52	8192	0	256	0	Y
AT89C55WD	20480	0	256	0	Y
AT89C51RC	32768	0	256	256	Y
AT89C51RB2	16384	0	256	1024	Y
AT89C51RC2	32768	0	256	1024	Y
AT89C51RD2	65536	0	256	1792	Y
AT89C51IC2	32768	0	256	1024	Y
AT89C51ID2	65536	2048	256	1792	Y
AT89C51ED2	65536	2048	256	1792	Y

Section 9

Instruction Set Reference

The following sections describe all the instructions supported by the Atmel® C51ASM assembler. These include the standard MCS®51 instructions, plus some additional extensions unique to the Atmel AT89LP Family of microcontrollers.

All instruction cycle counts are listed in units of system clocks. Most classic 8051s divide the XTAL clock by two to generate the system clock, therefore the listed number must be multiplied by two for 12-clocks per machine cycle devices. The listed number is equivalent to a device operating in X2 mode (6-clocks per machine cycle). Cycle counts are listed with three different qualifiers:

- **Classic**: Standard 8051 timing for the Atmel AT89C and Atmel AT89S families, or for Atmel AT89LP devices that are operating in a compatibility mode
- **Fast**: Timing for the Atmel AT89LP single-cycle family when not in compatibility mode
- **Compatibility**: Timing for the extended instructions of the AT89LP family during compatibility mode. These instructions are not available on “classic” device.

9.1 Addressing Modes

The instruction set supports the following addressing modes.

Table 9-1. 8051 Addressing Modes

Symbol	Description
R_n	Register R0-R7 of the currently selected Register Bank.
direct	8-bit internal data location address. This could be an internal data RAM location (0-127) or an SFR (128-255).
@R_i	8-bit internal data RAM location (0-255) addressed indirectly through the contents of R1 or R0.
#data	8-bit immediate constant included in instruction.
#data16	16-bit immediate constant included in instruction.
addr24	24-bit destination address used by EJMP or ECALL. A branch can be anywhere within the 16MB Program Memory address space.
addr16	16-bit destination address used by LJMP or LCALL. A branch can be anywhere within the 64KB Program Memory address space.
addr11	11-bit destination address used by AJMP or ACALL. A branch will be within the same 2KB segement of Program Memory as the first byte of the following instruction.
rel	Signed (two’s complement) 8-bit offset used by SJMP and all conditional jumps. Rnage is -128 to +127 bytes relative to the first byte of the following instruction.
bit	Directly addressed bit in internal data RAM or an SFR.

9.2 Instructions in Alphabetical Order

Table 9-2 lists the supported instructions of the Atmel® AT89 family of 8051 microcontrollers, organized by mnemonic.

Table 9-2. 8051 Instructions in Alphabetical Order

Mnemonic	Operands	Bytes	Hex Code	Page
ACALL	code addr	2	11, 31, 51, 71, 91, B1, D1, F1	9-6
ADD	A, #data	2	24	9-7
ADD	A, data addr	2	25	9-7
ADD	A, @Ri	1	26–27	9-7
ADD	A, Rn	1	28–2F	9-7
ADDC	A, #data	2	34	9-8
ADDC	A, data addr	2	35	9-8
ADDC	A, @Ri	1	36–37	9-8
ADDC	A, Rn	1	38–3F	9-8
AJMP	code addr	2	01, 21, 41, 61, 81, A1, C1, E1	9-9
ANL	A, #data	2	54	9-9
ANL	A, data addr	2	55	9-9
ANL	A, @Ri	1	56–57	9-9
ANL	A, Rn	1	58–5F	9-9
ANL	C, bit addr	2	82	9-11
ANL	C, /bit addr	2	B0	9-11
ANL	data addr, A	2	52	9-9
ANL	data addr, #data	3	53	9-9
ASR	M	2	A5 03	9-11
BREAK		2	A5 00	9-12
CJNE	A, #data, code addr	3	B4	9-12
CJNE	A, data addr, code addr	3	B5	9-12
CJNE	@Ri, #data, code addr	3	B6–B7	9-12
CJNE	Rn, #data, code addr	3	B8–BF	9-12
CJNE	A, @Ri, code addr	3	A5 B6, A5 B7	9-12
CLR	A	1	E4	9-14
CLR	bit addr	2	C2	9-15
CLR	C	1	C3	9-15
CLR	M	2	A5 E4	9-14
CPL	A	1	F4	9-15
CPL	bit addr	2	B2	9-16
CPL	C	1	B3	9-16
DA	A	1	D4	9-16
DEC	A	1	14	9-17
DEC	data addr	2	15	9-17
DEC	@Ri	1	16–17	9-17

Table 9-2. 8051 Instructions in Alphabetical Order

Mnemonic	Operands	Bytes	Hex Code	Page
DEC	Rn	1	18–1F	9-17
DIV	AB	1	84	9-18
DJNZ	data addr, code addr	3	D5	9-19
DJNZ	Rn, code addr	2	D8–DF	9-19
ECALL	code addr	5	A5 12	9-20
EJMP	code addr	5	A5 02	9-21
ERET		2	A5 22	9-21
INC	A	1	04	9-22
INC	data addr	2	05	9-22
INC	@Ri	1	06–07	9-22
INC	Rn	1	08–0F	9-22
INC	DPTR	1	A3	9-23
INC	/DPTR	2	A5 A3	9-23
JB	bit addr, code addr	3	20	9-24
JBC	bit addr, code addr	3	10	9-24
JC	code addr	2	40	9-25
JMP	@A+DPTR	1	73	9-25
JMP	@A+PC	2	A5 73	9-25
JNB	bit addr, code addr	3	30	9-26
JNC	code addr	2	50	9-26
JNZ	code addr	2	70	9-27
JZ	code addr	2	60	9-27
LCALL	code addr	3	12	9-28
LJMP	code addr	3	02	9-28
LSL	M	2	A5 23	9-29
MAC	AB	2	A5 A4	9-29
MOV	A, #data	2	74	9-30
MOV	A, data addr	2	E5	9-30
MOV	A, @Ri	1	E6–E7	9-30
MOV	A, Rn	1	E8–EF	9-30
MOV	data addr, #data	3	75	9-30
MOV	@Ri, #data	2	76–77	9-30
MOV	Rn, #data	2	78–7F	9-30
MOV	data addr, data addr	3	85	9-30
MOV	data addr, @Ri	2	86–87	9-30
MOV	data addr, Rn	2	88–8F	9-30
MOV	DPTR, #data	3	90	9-33
MOV	/DPTR, #data	4	A5 90	9-33
MOV	bit addr, C	2	92	9-34

Table 9-2. 8051 Instructions in Alphabetical Order

Mnemonic	Operands	Bytes	Hex Code	Page
MOV	C, bit addr	2	A2	9-34
MOV	@Ri, data addr	2	A6–A7	9-30
MOV	Rn, data addr	2	A8–AF	9-30
MOV	data addr, A	2	F5	9-30
MOV	@Ri, A	1	F6–F7	9-30
MOV	Rn, A	1	F8–FF	9-30
MOVC	A, @A+DPTR	1	93	9-34
MOVC	A, @A+/DPTR	2	A5 93	9-34
MOVC	A, @A+PC	1	83	9-34
MOVX	A, @DPTR	1	E0	9-35
MOVX	A, @/DPTR	2	A5 E0	9-35
MOVX	A, @Ri	1	E2–E3	9-35
MOVX	@DPTR, A	1	F0	9-35
MOVX	@/DPTR, A	2	A5 F0	9-35
MOVX	@Ri, A	1	F2–F3	9-35
MUL	AB	1	A4	9-37
NOP		1	00	9-38
ORL	A, #data	2	44	9-38
ORL	A, data addr	2	45	9-38
ORL	A, @Ri	1	46–47	9-38
ORL	A, Rn	1	48–4F	9-38
ORL	C, bit addr	2	72	9-40
ORL	C, /bit addr	2	A0	9-40
ORL	data addr, A	2	42	9-38
ORL	data addr, #data	3	43	9-38
POP	data addr	2	D0	9-40
PUSH	data addr	2	C0	9-41
RET		1	22	9-41
RETI		1	32	9-42
RL	A	1	23	9-43
RLC	A	1	33	9-44
RR	A	1	03	9-44
RRC	A	1	13	9-45
SETB	bit addr	2	D2	9-45
SETB	C	1	D3	9-45
SJMP	code addr	2	80	9-46
SUBB	A, #data	2	94	9-46
SUBB	A, data addr	2	95	9-46
SUBB	A, @Ri	1	96–97	9-46

Table 9-2. 8051 Instructions in Alphabetical Order

Mnemonic	Operands	Bytes	Hex Code	Page
SUBB	A, Rn	1	98–9F	9-46
SWAP	A	1	C4	9-47
XCH	A, data addr	2	C5	9-48
XCH	A, @Ri	1	C6–C7	9-48
XCH	A, Rn	1	C8–CF	9-48
XCHD	A, @Ri	1	D6–D7	9-48
XRL	A, #data	2	64	9-49
XRL	A, data addr	2	65	9-49
XRL	A, @Ri	1	66–67	9-49
XRL	A, Rn	1	68–6F	9-49
XRL	data addr, A	2	62	9-49
XRL	data addr, #data	3	63	9-49

9.3 Instruction Set Extensions

The following instructions are extensions to the standard 8051 instruction set that provide enhanced capabilities not found in standard 8051 devices. All extended instructions start with an A5H escape code. For this reason random A5H reserved codes should not be placed in the instruction stream even though other devices may have treated these as NOPs.

These instructions are only supported on Atmel® AT89LP devices and individual AT89LP devices may not support all of these instructions. For instruction cycle times, the number of internal CPU clock cycles is listed. The classic 80C51 compatible timing also lists the number of machine cycles in parantheses.

Table 9-3. Instruction Set Extensions

Hex Code	Bytes	Mnemonic	Operands	Cycles		Device Support						
				Classic	Fast	AT89LP213	AT89LP214	AT89LP216	AT89LP428	AT89LP52	AT89LP828	AT89LP6440
A5 00	2	BREAK		12 (2)	2	X	X	X	X		X	X
A5 02	5	EJMP	addr24	18 (3)	6							
A5 03	2	ASR	M	12 (2)	2							X
A5 12	5	ECALL	addr24	18 (3)	6							
A5 22	2	ERET		18 (3)	6							
A5 23	2	LSL	M	12 (2)	2							X
A5 73	2	JMP	@A+PC	18 (3)	3	X	X	X	X	X	X	X
A5 90	4	MOV	/DPTR, #data16	18 (3)	4				X	X	X	X
A5 93	2	MOVC	A, @A+/DPTR	18 (3)	4				X	X	X	X
A5 A3	2	INC	/DPTR	18 (3)	3				X	X	X	X
A5 A4	2	MAC	AB	30 (5)	9							X
A5 B6	3	CJNE	A, @R0, rel	18 (3)	4				X	X	X	X

Table 9-3. Instruction Set Extensions

Hex Code	Bytes	Mnemonic	Operands	Cycles		Device Support						
				Classic	Fast	AT89LP213	AT89LP214	AT89LP216	AT89LP428	AT89LP52	AT89LP828	AT89LP6440
A5 B7	3	CJNE	A, @R1, rel	18 (3)	4				X	X	X	X
A5 E0	2	MOVX	A, @/DPTR	18 (3)	3/5				X	X	X	X
A5 E4	2	CLR	M	12 (2)	2							X
A5 F0	2	MOVX	@DPTR, A	18 (3)	3/5				X	X	X	X

9.4 Instruction Definitions

Detailed descriptions of all the instructions supported by the Atmel® C51ASM assembler are listed below. All stack-realted instructions include entries for both the standard IDATA stack and the extended EDATA stack.

9.4.1 ACALL addr11

Function: Absolute Call

Description: ACALL unconditionally calls a subroutine located at the indicated address. The instruction increments the PC twice to obtain the address of the following instruction, then pushes the 16-bit result onto the stack (low-order byte first) and increments the Stack Pointer twice. The destination address is obtained by successively concatenating the five high-order bits of the incremented PC, opcode bits 7 through 5, and the second byte of the instruction. The subroutine called must therefore start within the same 2 K block of the program memory as the first byte of the instruction following ACALL. No flags are affected.

Example: Initially SP equals 07H. The label SUBRTN is at program memory location 0345 H. After executing the following instruction,

```
ACALL   SUBRTN
```

at location 0123H, SP contains 09H, internal RAM locations 08H and 09H will contain 25H and 01H, respectively, and the PC contains 0345H.

ACALL addr11 (Standard Stack)

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

3 Fast Mode

Encoding:

a10	a9	a8	1	0	0	0	1	a7	a6	a5	a4	a3	a2	a1	a0

Operation: ACALL
(PC) ←(PC) + 2
(SP_{7-0}) ←(SP_{7-0}) + 1
((SP_{7-0})) ←(PC_{7-0})
(SP_{7-0}) ←(SP_{7-0}) + 1
((SP_{7-0})) ←(PC_{15-8})
(PC_{10-0}) ←page address

9.4.1 ACALL addr11

ACALL addr11 (Extended Stack)

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

5 Fast Mode

Encoding:	a10	a9	a8	1	0	0	0	1	a7	a6	a5	a4	a3	a2	a1	a0

Operation: ACALL (SP_{15-8} = SPX)
(PC) ←(PC) + 2
(SP_{15-0}) ←(SP_{15-0}) + 1
((SP_{15-0})) ←(PC_{7-0})
(SP_{15-0}) ←(SP_{15-0}) + 1
((SP_{15-0})) ←(PC_{15-8})
(PC_{10-0}) ←page address

9.4.2 ADD A, <src-byte>

Function: Add

Description: ADD adds the byte variable indicated to the Accumulator, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not bit 6; otherwise, OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The Accumulator holds 0C3H (1100001IB), and register 0 holds 0AAH (10101010B). The following instruction,

```
ADD  A,R0
```

leaves 6DH (01101101B) in the Accumulator with the AC flag cleared and both the carry flag and OV set to 1.

ADD A,R_n

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	0	0	1	0	1	r	r	r

Operation: (A) ←(A) + (R_n)

ADD A,direct

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	0	1	0	0	1	0	1	direct address

Operation: (A) ←(A) + (direct)

ADD A,@R_i

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	0	1	0	0	1	1	i

Operation: (A) ←(A) + ((R_i))

9.4.2 ADD A, <src-byte>

ADD A,#data

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	0	1	0	0	1	0	0	immediate data

Operation: ADD
(A) ←(A) + #data

9.4.3 ADDC A, <src-byte>

Function: Add with Carry

Description: ADDC simultaneously adds the byte variable indicated, the carry flag and the Accumulator contents, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The Accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) with the carry flag set. The following instruction,

```
ADDC A,R0
```

leaves 6EH (01101110B) in the Accumulator with AC cleared and both the Carry flag and OV set to 1.

ADDC A,R_n

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	0	0	1	1	1	r	r	r

Operation: ADDC
(A) ←(A) + (C) + (R_n)

ADDC A,direct

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	0	1	1	0	1	0	1	direct address

Operation: ADDC
(A) ←(A) + (C) + (direct)

ADDC A,@R_i

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	0	1	1	0	1	1	i

Operation: ADDC
(A) ←(A) + (C) + ((R_i))

9.4.3 ADDC A, <src-byte>

ADDC A,#data

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	0	1	1	0	1	0	0	immediate data

Operation: ADDC
(A) ←(A) + (C) + #data

9.4.4 AJMP addr11

Function: Absolute Jump

Description: AJMP transfers program execution to the indicated address, which is formed at run-time by concatenating the high-order five bits of the PC (after incrementing the PC twice), opcode bits 7 through 5, and the second byte of the instruction. The destination must therfore be within the same 2 K block of program memory as the first byte of the instruction following AJMP.

Example: The label JMPADR is at program memory location 0123H. The following instruction,
AJMP JMPADR
is at location 0345H and loads the PC with 0123H.

AJMP addr11

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

3 Fast Mode

Encoding:	a10	a9	a8	0	0	0	0	1	a7	a6	a5	a4	a3	a2	a1	a0

Operation: AJMP
(PC) ←(PC) + 2
(PC_{10-0}) ←page address

9.4.5 ANL <dest-byte>, <src-byte>

Function: Logical-AND for byte variables

Description: ANL performs the bitwise logical-AND operation between the variables indicated and stores the results in the destination variable. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

Example: If the Accumulator holds 0C3H (1100001lB), and register 0 holds 55H (01010101B), then the following instruction,

ANL A,R0

leaves 41H (01000001B) in the Accumulator.

When the destination is a directly addressed byte, this instruction clears combinations of bits in any RAM location or hardware register. The mask byte determining the pattern of bits to be cleared would either be a constant contained in the instruction or a value computed in the Accumulator at run-time. The following instruction,

ANL P1,#01110011B

clears bits 7, 3, and 2 of output port 1.

9.4.5 ANL <dest-byte>, <src-byte>

ANL A,R_n

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	0	1	0	1	1	r	r	r

Operation: (A) ←(A) ∧ (R_n)

ANL A,direct

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	1	0	1	0	1	0	1	direct address

Operation: (A) ←(A) ∧ (direct)

ANL A,@R_i

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle

2 Fast Mode

Encoding:	0	1	0	1	0	1	1	i

Operation: (A) ←(A) ∧ ((R_i))

ANL A,#data

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	1	0	1	0	1	0	0	immediate data

Operation: (A) ←(A) ∧ #data

ANL direct,A

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	1	0	1	0	0	1	0	direct address

Operation: (direct) ←(direct) ∧ (A)

ANL direct,#data

Bytes: 3

Cycles: 12 Classic (2 Machine Cycles)

3 Fast Mode

Encoding:	0	1	0	1	0	0	1	1	direct address	immediate data

Operation: (direct) ←(direct) ∧ #data

9.4.6 ANL C,<src-bit>

Function: Logical-AND for bit variables

Description: If the Boolean value of the source bit is a logical 0, then ANL C clears the carry flag; otherwise, this instruction leaves the carry flag in its current state. A slash (/) preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, *but the source bit itself is not affected*. No other flags are affected.

Only direct addressing is allowed for the source operand.

Example: Set the carry flag if, and only if, P1.0 = 1, ACC.7 = 1, and OV = 0:

```
MOV     C,P1.0    ..... ..... ;LOAD CARRY WITH INPUT PIN STATE
ANL     C,ACC.7 ..... ..... ;AND CARRY WITH ACCUM. BIT 7
ANL     C,/OV     ..... ..... ;AND WITH INVERSE OF OVERFLOW FLAG
```

ANL C,bit

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

2 Fast Mode

Encoding:

1	0	0	0	0	0	1	0	bit address

Operation: ANL

(C) ←(C) ∧ (bit)

ANL C,/bit

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

2 Fast Mode

Encoding:

1	0	1	1	0	0	0	0	bit address

Operation: ANL

(C) ←(C) ∧ (bit)

9.4.7 ASR M

Function: Shift MAC Accumulator Right Arithmetically

Description: The forty bits in the M register are shifted one bit to the right. Bit 39 retains its value to preserve the sign of the value. No flags are affected.

Example: The M register holds the value 0C5B1A29384H . The following instruction,

```
ASR   M
```

leaves the M register holding the value 0E2D8D149C2H.

Bytes: 2

Cycles: 12 Compatibility (2 Machine Cycles)

2 Fast Mode

Encoding:

A5		0	0	0	0	0	0	1	1

Operation: ASR

$(M_n) \leftarrow (M_{n+1})$ n = 0 - 38

$(M_{39}) \leftarrow (M_{39})$

9.4.8 BREAK

Function: Software Breakpoint (Halt execution)

Description: BREAK transfers control from normal execution to the On-Chip Debug (OCD) handler if OCD is enabled. The PC is left pointing to the following instruction. If OCD is disabled, BREAK acts as a double NOP. No flags are affected.

Example: If On-Chip Debugging is allowed, the following instruction,
BREAK
will halt instruction execution prior to the immediately following instruction. If debugging is not allowed, the BREAK is treated as a double NOP.

Bytes: 2

Cycles: 12 Compatibility (2 Machine Cycles)

2 Fast Mode

Encoding:

A5	0	0	0	0	0	0	0	0

Operation: (PC) ←(PC) + 2

9.4.9 CJNE <dest-byte>,<src-byte>, rel

Function: Compare and Jump if Not Equal.

Description: CJNE compares the magnitudes of the first two operands and branches if their values are not equal. The branch destination is computed by adding the signed relative-displacement in the last instruction byte to the PC, after incrementing the PC to the start of the next instruction. The carry flag is set if the unsigned integer value of <dest-byte> is less than the unsigned integer value of <src-byte>; otherwise, the carry is cleared. Neither operand is affected.

The first two operands allow four addressing mode combinations: the Accumulator may be compared with any directly addressed byte or immediate data, and any indirect RAM location or working register can be compared with an immediate constant.

Example: The Accumulator contains 34H. Register 7 contains 56H. The first instruction in the sequence,

```
                    ..... CJNE .... .....R7, # 60H, NOT_EQ
;                   ..... . . . ..... .....  . . . .  ;R7 = 60H.
NOT_EQ:             .....JC  ..... .....REQ_LOW;IF R7 < 60H.
;                   ..... . . ..... .....  . . . .  ;R7 > 60H.
```

sets the carry flag and branches to the instruction at label NOT_EQ. By testing the carry flag, this instruction determines whether R7 is greater or less than 60H.

If the data being presented to Port 1 is also 34H, then the following instruction,

WAIT:CJNE A, P1,WAIT

clears the carry flag and continues with the next instruction in sequence, since the Accumulator does equal the data read from P1. (If some other value was being input on P1, the program loops at this point until the P1 data changes to 34H.)

CJNE A,direct,rel

Bytes: 3

Cycles: 12 Classic (2 Machine Cycles)

Cycles: 4 Fast Mode

Encoding:

1	0	1	1	0	1	0	1	direct address	rel. address

Operation:
(PC) ←(PC) + 3
IF (A) < > (*direct*) THEN
  (PC) ←(PC) + *relative offset*
IF (A) < (*direct*) THEN
  (C) ←1
ELSE
  (C) ←0

9.4.9 CJNE <dest-byte>,<src-byte>, rel

CJNE A,#data,rel

Bytes: 3

Cycles: 12 Classic (2 Machine Cycles)

Cycles: 4 Fast Mode

Encoding:	1	0	1	1	0	1	0	0	immediate data	rel. address

Operation: (PC) ←(PC) + 3
IF (A) < > *data*
THEN
(PC) ←(PC) + *relative offset*
IF (A) < *data*
THEN
(C) ←1
ELSE
(C) ←0

CJNE R_n,#data,rel

Bytes: 3

Cycles: 12 Classic (2 machine Cycles)

4 Fast Mode

Encoding:	1	0	1	1	1	r	r	r	immediate data	rel. address

Operation: (PC) ←(PC) + 3
IF (R_n) < > *data*
THEN
(PC) ←(PC) + *relative offset*
IF (R_n) < *data*
THEN
(C) ←1
ELSE
(C) ←0

CJNE @R_i,data,rel

Bytes: 3

Cycles: 12 Classic (2 machine Cycles)

4 Fast Mode

Encoding:	1	0	1	1	0	1	1	i	immediate data	rel. address

Operation: (PC) ←(PC) + 3
IF ((R_i)) < > *data*
THEN
(PC) ←(PC) + *relative offset*
IF ((R_i)) < *data*
THEN
(C) ←1
ELSE
(C) ←0

9.4.9 CJNE <dest-byte>,<src-byte>, rel

CJNE A, @R_i,,rel

Bytes: 3

Cycles: 12 Compatibility (2 machine Cycles)

4 Fast Mode

Encoding:

A5	1	0	1	1	0	1	1	i	rel. address

Operation: CJNE
(PC) ← (PC) + 3
IF (A) ≠ ((R_i))
THEN
    (PC) ← (PC) + relative offset
IF (A) < ((R_i))
THEN
    (C) ← 1
ELSE
    (C) ← 0

9.4.10 CLR A

Function: Clear Accumulator

Description: CLR A clears the Accumulator (all bits set to 0). No flags are affected

Example: The Accumulator contains 5CH (01011100B). The following instruction,CLR Aleaves the Accumulator set to 00H (00000000B).

CLR A

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:

1	1	1	0	0	1	0	0

Operation: CLR
(A) ←0

9.4.11 CLR M

Function: Clear MAC Accumulator

Description: CLR M clears the 40-bit M register. No flags are affected.

Example: The M registercontains 123456789AH. The following instruction,

CLR M

leaves the M register set to 0000000000H.

Bytes: 2

Cycles: 6 Compatibility (1 Machine Cycle)

2 Fast Mode

Encoding:

A5	1	1	1	0	0	1	0	0

Operation: (M) ←0

9.4.12 CLR bit

Function: Clear bit

Description: CLR bit clears the indicated bit (reset to 0). No other flags are affected. CLR can operate on the carry flag or any directly addressable bit.

Example: Port 1 has previously been written with 5DH (01011101B). The following instruction,CLR P1.2 leaves the port set to 59H (01011001B).

CLR C

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	1	1	0	0	0	0	1	1

Operation: CLR
(C) ←0

CLR bit

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	1	1	0	0	0	0	1	0	bit address

Operation: (bit) ←0

9.4.13 CPL A

Function: Complement Accumulator

Description: CPLA logically complements each bit of the Accumulator (one's complement). Bits which previously contained a 1 are changed to a 0 and vice-versa. No flags are affected.

Example: The Accumulator contains 5CH (01011100B). The following instruction,

```
CPL     A
```

leaves the Accumulator set to 0A3H (10100011B).

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	1	1	1	1	0	1	0	0

Operation: (A) ← (A)

9.4.14 CPL bit

Function: Complement bit

Description: CPL bit complements the bit variable specified. A bit that had been a 1 is changed to 0 and vice-versa. No other flags are affected. CLR can operate on the carry or any directly addressable bit.

Note: When this instruction is used to modify an output pin, the value used as the original data is read from the output data latch, *not* the input pin.

Example: Port 1 has previously been written with 5BH (01011101B). The following instruction sequence,CPL P1.1CPL P1.2 leaves the port set to 5BH (01011011B).

CPL C

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	1	0	1	1	0	0	1	1

Operation: (C) ← (C)

CPL bit

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	1	0	1	1	0	0	1	0	bit address

Operation: (bit) ← (bit)

9.4.15 DA A

Function: Decimal-adjust Accumulator for Addition

Description: DA A adjusts the eight-bit value in the Accumulator resulting from the earlier addition of two variables (each in packed-BCD format), producing two four-bit digits. Any ADD or ADDC instruction may have been used to perform the addition.

If Accumulator bits 3 through 0 are greater than nine (xxxx1010-xxxx1111), or if the AC flag is one, six is added to the Accumulator producing the proper BCD digit in the low-order nibble. This internal addition sets the carry flag if a carry-out of the low-order four-bit field propagates through all high-order bits, but it does not clear the carry flag otherwise.

If the carry flag is now set, or if the four high-order bits now exceed nine (1010xxxx-1111xxxx), these high-order bits are incremented by six, producing the proper BCD digit in the high-order nibble. Again, this sets the carry flag if there is a carry-out of the high-order bits, but does not clear the carry. The carry flag thus indicates if the sum of the original two BCD variables is greater than 100, allowing multiple precision decimal addition. OV is not affected.

All of this occurs during the one instruction cycle. Essentially, this instruction performs the decimal conversion by adding 00H, 06H, 60H, or 66H to the Accumulator, depending on initial Accumulator and PSW conditions.

Note: DA A *cannot* simply convert a hexadecimal number in the Accumulator to BCD notation, nor does DAA apply to decimal subtraction.

9.4.15 DA A

Example: The Accumulator holds the value 56H (01010110B), representing the packed BCD digits of the decimal number 56. Register 3 contains the value 67H (01100111B), representing the packed BCD digits of the decimal number 67. The carry flag is set. The following instruction sequence

```
ADDC   A,R3
DA     A
```

first performs a standard two's-complement binary addition, resulting in the value 0BEH (10111110) in the Accumulator. The carry and auxiliary carry flags are cleared.

The Decimal Adjust instruction then alters the Accumulator to the value 24H (00100100B), indicating the packed BCD digits of the decimal number 24, the low-order two digits of the decimal sum of 56, 67, and the carry-in. The carry flag is set by the Decimal Adjust instruction, indicating that a decimal overflow occurred. The true sum of 56, 67, and 1 is 124.

BCD variables can be incremented or decremented by adding 01H or 99H. If the Accumulator initially holds 30H (representing the digits of 30 decimal), then the following instruction sequence,

```
ADD    A, # 99H
DA     A
```

leaves the carry set and 29H in the Accumulator, since 30 + 99 = 129. The low-order byte of the sum can be interpreted to mean 30 - 1 = 29.

DA A

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:

1	1	0	1	0	1	0	0

Operation: -contents of Accumulator are BCD

IF [[(A_{3-0}) > 9] ∨ [(AC) = 1]]
THEN $(A_{3-0}) \leftarrow (A_{3-0}) + 6$
AND
IF [[(A_{7-4}) > 9] ∨ [(C) = 1]]
THEN $(A_{7-4}) \leftarrow (A_{7-4}) + 6$

9.4.16 DEC <byte>

Function: Decrement

Description: DEC byte decrements the variable indicated by 1. An original value of 00H underflows to 0FFH. No flags are affected. Four operand addressing modes are allowed: accumulator, register, direct, or register-indirect.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

Example: Register 0 contains 7FH (01111111B). Internal RAM locations 7EH and 7FH contain 00H and 40H, respectively. The following instruction sequence,

```
DEC    @R0
DEC    R0
DEC    @R0
```

leaves register 0 set to 7EH and internal RAM locations 7EH and 7FH set to 0FFH and 3FH.

DEC A

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:

0	0	0	1	0	1	0	0

Operation: DEC
(A) ←(A) - 1

9.4.16 DEC <byte>

DEC R_n

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	0	0	0	1	1	r	r	r

Operation: DEC
$(R_n) \leftarrow (R_n) - 1$

DEC direct

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	0	0	1	0	1	0	1	direct address

Operation: DEC
(direct) ←(direct) - 1

DEC @R_i

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	0	0	1	0	1	1	i

Operation: DEC
$((R_i)) \leftarrow ((R_i)) - 1$

9.4.17 DIV AB

Function: Divide

Description: DIV AB divides the unsigned eight-bit integer in the Accumulator by the unsigned eight-bit integer in register B. The Accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags are cleared.

Exception: if B had originally contained 00H, the values returned in the Accumulator and B-register are undefined and the overflow flag are set. The carry flag is cleared in any case.

Example: The Accumulator contains 251 (0FBH or 11111011B) and B contains 18 (12H or 00010010B). The following instruction,

```
DIV     AB
```

leaves 13 in the Accumulator (0DH or 00001101B) and the value 17 (11H or 00010001B) in B, since 251 = (13 x 18) + 17. Carry and OV are both cleared.

DIV AB

Bytes: 1

Cycles: 24 Classic (4 Machine Cycles)

4 Fast Mode

Encoding:	1	0	0	0	0	1	0	0

Operation: DIV
$(A)_{15-8} \leftarrow (A)/(B)$
$(B)_{7-0}$

9.4.18 DJNZ <byte>,<rel-addr>

Function: Decrement and Jump if Not Zero

Description: DJNZ decrements the location indicated by 1, and branches to the address indicated by the second operand if the resulting value is not zero. An original value of 00H underflows to 0FFH. No flags are affected. The branch destination is computed by adding the signed relative-displacement value in the last instruction byte to the PC, after incrementing the PC to the first byte of the following instruction.

The location decremented may be a register or directly addressed byte.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

Example: Internal RAM locations 40H, 50H, and 60H contain the values 01H, 70H, and 15H, respectively. The following instruction sequence,

```
DJNZ    40H,LABEL_1
DJNZ    50H,LABEL_2
DJNZ    60H,LABEL_3
```

causes a jump to the instruction at label LABEL_2 with the values 00H, 6FH, and 15H in the three RAM locations. The first jump was *not* taken because the result was zero.

This instruction provides a simple way to execute a program loop a given number of times or for adding a moderate time delay (from 2 to 512 machine cycles) with a single instruction. The following instruction sequence,

```
        MOV     ..... R2, # 8
TOGGLE: CPL     ..... P1.7
        DJNZ    ..... R2,TOGGLE
```

toggles P1.7 eight times, causing four output pulses to appear at bit 7 of output Port 1. Each pulse lasts three machine cycles; two for DJNZ and one to alter the pin.

DJNZ R_n,rel

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

3 Fast Mode

Encoding:

1	1	0	1	1	r	r	r	rel. address	

Operation:
(PC) ←(PC) + 2
(R_n) ←(R_n) - 1
IF (R_n) > 0 or (R_n) < 0
THEN
(PC) ←(PC) + rel

DJNZ direct,rel

Bytes: 3

Cycles: 12 Classic (2 Machine Cycles)

4 Fast Mode

Encoding:

1	1	0	1	0	1	0	1	direct address	rel. address

Operation:
(PC) ←(PC) + 2
(direct) ←(direct) - 1
IF (direct) > 0 or (direct) < 0
THEN
(PC) ←(PC) + rel

9.4.19 ECALL addr24

Function: Extended call

Description: ECALL calls a subroutine located at the indicated address. The instruction adds five to the program counter to generate the address of the next instruction and then pushes the 24-bit result onto the stack (low byte first), incrementing the Stack Pointer by three. The high-order middle, and low-order bytes of the PC are then loaded, respectively, with the third, fourth, and fifth bytes of the ECALL instruction. Program execution continues with the instruction at this address. The subroutine may therefore begin anywhere in the 16MB program memory address space. No flags are affected.

ECALL is only supported on devices with more than 64KB of program memory.

Example: Initially the Stack Pointer equals 07H. The label SUBRTN is assigned to program memory location 123456H. After executing the instruction,

ECALL SUBRTN

at location 012345H, the Stack Pointer will contain 0A9H, internal RAM locations 08H, 09H and 0AH will contain 4AH, 23H and 01H, and the PC will contain 123456H.

ECALL addr24 (Standard Stack)

Bytes: 5

Cycles: 18 Classic (3 Machine Cycles)

6 Fast Mode

Encoding:

A5	12	addr23-addr16	addr15-addr8	addr7-addr0

Operation: ECALL

$(PC) \leftarrow (PC) + 5$
$(SP_{7-0}) \leftarrow (SP_{7-0}) + 1$
$((SP_{7-0})) \leftarrow (PC_{7-0})$
$(SP_{7-0}) \leftarrow (SP_{7-0}) + 1$
$((SP_{7-0})) \leftarrow (PC_{15-8})$
$(SP_{7-0}) \leftarrow (SP_{7-0}) + 1$
$((SP_{7-0})) \leftarrow (PC_{23-16})$
$(PC) \leftarrow addr_{23-0}$

ECALL addr24 (Extended Stack)

Bytes: 5

Cycles: 18 Classic (3 Machine Cycles)

8 Fast Mode

Encoding:

A5	12	addr23-addr16	addr15-addr8	addr7-addr0

Operation: ECALL ($SP_{15-}8 = SPX$)

$(PC) \leftarrow (PC) + 5$
$(SP_{15-0}) \leftarrow (SP_{15-0}) + 1$
$((SP_{15-0})) \leftarrow (PC_{7-0})$
$(SP_{15-0}) \leftarrow (SP_{15-0}) + 1$
$((SP_{15-0})) \leftarrow (PC_{15-8})$
$(SP_{15-0}) \leftarrow (SP_{15-0}) + 1$
$((SP_{15-0})) \leftarrow (PC_{23-16})$
$(PC) \leftarrow addr_{23-0}$

9.4.20 EJMP addr24

Function: Extended jump

Description: EJMP causes an unconditional branch to the indicated address, by loading the high-order, middle, and low-order bytes of the PC (respectively) with the third, fourth, and fifth instruction bytes. The destination may therefore be anywhere in the 16MB program memory address space. No flags are affected.

EJMP is only supported on devices with more than 64KB of program memory.

Example: The label JMPADR is assigned to the instruction at program memory location 123456H. The instruction,

```
EJMP    JMPADR
```

at location 0123H will load the program counter with 123456H.

EJMP addr24

Bytes: 5

Cycles: 18 Compatibility (3 Machine Cycles)

6 Fast Mode

Encoding:

A5	12	addr23-addr16	addr15-addr8	addr7-addr0

Operation: EJMP

$(PC) \leftarrow addr_{23-0}$

9.4.21 ERET

Function: Extended Return from subroutine

Description: ERET pops the high-, mniddle and low-order bytes of the PC successively from the stack, decrementing the Stack Pointer by three. Program execution continues at the resulting address, generally the instruction immediately following an ECALL. No flags are affected.

ERET is only supported on devices with more than 64KB of program memory.

Example: The Stack Pointer originally contains the value 0BH. Internal RAM locations 09H, 0AH and 0BH contain the values 45H, 23H and 01H, respectively. The following instruction,

```
ERET
```

leaves the Stack Pointer equal to the value 08H. Program execution continues at location 012345H.

ERET (Standard Stack)

Bytes: 1

Cycles: 18 Compatibility (3 Machine Cycles)

5 Fast Mode

Encoding:

1	0	1	0	0	1	0	1	0	0	1	0	0	0	1	0

Operation: RET

$(PC_{23-16}) \leftarrow((SP_{7-0}))$
$(SP_{7-0}) \leftarrow(SP_{7-0}) - 1$
$(PC_{15-8}) \leftarrow((SP_{7-0}))$
$(SP_{7-0}) \leftarrow(SP_{7-0}) - 1$
$(PC_{7-0}) \leftarrow((SP_{7-0}))$
$(SP_{7-0}) \leftarrow(SP_{7-0}) - 1$

9.4.21 ERET

ERET (Extended Stack)

Bytes: 1

Cycles: 18 Compatibility (3 Machine Cycles)

8 Fast Mode

Encoding:	1	0	1	0	0	1	0	1	0	0	1	0	0	0	1	0

Operation: RET (SP_{15-8} = SPX)
$(PC_{23-16}) \leftarrow((SP_{15-0}))$
$(SP_{15-0}) \leftarrow(SP_{15-0}) - 1$
$(PC_{15-8}) \leftarrow((SP_{15-0}))$
$(SP_{15-0}) \leftarrow(SP_{15-0}) - 1$
$(PC_{7-0}) \leftarrow((SP_{15-0}))$
$(SP_{15-0}) \leftarrow(SP_{15-0}) - 1$

9.4.22 INC <byte>

Function: Increment

Description: INC increments the indicated variable by 1. An original value of 0FFH overflows to 00H. No flags are affected. Three addressing modes are allowed: register, direct, or register-indirect.

Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, *not* the input pins.

Example: Register 0 contains 7EH (011111110B). Internal RAM locations 7EH and 7FH contain 0FFH and 40H, respectively. The following instruction sequence,

```
INC     @R0
INC      R0
INC     @R0
```

leaves register 0 set to 7FH and internal RAM locations 7EH and 7FH holding 00H and 41H, respectively.

INC A

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	0	0	0	0	0	1	0	0

Operation: (A) ←(A) + 1

INC R_n

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	0	0	0	0	1	r	r	r

Operation: $(R_n) \leftarrow(R_n) + 1$

INC direct

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	0	0	0	0	1	0	1	direct address

Operation: (direct) ←(direct) + 1

9.4.22 INC <byte>

INC @R_i

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	0	0	0	0	1	1	i

Operation: INC
$((R_i)) \leftarrow((R_i)) + 1$

9.4.23 INC DPTR

Function: Increment Data Pointer

Description: INC DPTR increments the 16-bit data pointer by 1. A 16-bit increment (modulo 2^{16}) is performed, and an overflow of the low-order byte of the data pointer (DPL) from 0FFH to 00H increments the high-order byte (DPH). No flags are affected. For device with two data pointers. The data pointer to be incremented is slected by DPS usually in the AUXR1 register. This is the only 16-bit register which can be incremented. INC /DPTR is only available on devices with two or more data pointers.

Example: Registers DPH and DPL contain 12H and 0FEH, respectively. The following instruction sequence,

```
INC     DPTR
INC     DPTR
INC     DPTR
```

changes DPH and DPL to 13H and 01H.

INC DPTR

Bytes: 1

Cycles: 12 Classic (2 Machine Cycles)

2 Fast Mode

Encoding:	1	0	1	0	0	0	1	1

Operation:

```
INC
IF DPS = 0
THEN
  (DPTR0) ←(DPTR0) + 1
ELSE
  (DPTR1) ←(DPTR1) + 1
```

INC /DPTR

Bytes: 2

Cycles: 18 Compatibility (3 Machine Cycles)

3 Fast Mode

Encoding:	1	0	1	0	0	1	0	1	1	0	1	0	0	0	1	1

Operation:

```
INC
IF DPS = 0
THEN
  (DPTR1) ←(DPTR1) + 1
ELSE
  (DPTR0) ←(DPTR0) + 1
```

9.4.24 JB bit,rel

Function: Jump if Bit set

Description: If the indicated bit is a one, JB jump to the address indicated; otherwise, it proceeds with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. The bit tested is not modified. No flags are affected.

Example: The data present at input port 1 is 11001010B. The Accumulator holds 56 (01010110B). The following instruction sequence,

```
JB      P1.2,LABEL1
JB      ACC. 2,LABEL2
```

causes program execution to branch to the instruction at label LABEL2.

JB bit,rel

Bytes: 3

Cycles: 12 Classic (2 Machine Cycles)

4 Fast Mode

Encoding:

0	0	1	0	0	0	0	0	bit address	rel. address

Operation:

```
JB
(PC) ←(PC) + 3
IF (bit) = 1
THEN
    (PC) ←(PC) + rel
```

9.4.25 JBC bit,rel

Function: Jump if Bit is set and Clear bit

Description: If the indicated bit is one, JBC branches to the address indicated; otherwise, it proceeds with the next instruction. *The bit will not be cleared if it is already a zero.* The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. No flags are affected.

Note: When this instruction is used to test an output pin, the value used as the original data will be read from the output data latch, *not* the input pin.

Example: The Accumulator holds 56H (01010110B). The following instruction sequence,

```
JBC     ACC.3,LABEL1
JBC     ACC.2,LABEL2
```

causes program execution to continue at the instruction identified by the label LABEL2, with the Accumulator modified to 52H (01010010B).

JBC bit,rel

Bytes: 3

Cycles: 12 Classic (2 Machien Cycles)

4 Fast Mode

Encoding:

0	0	0	1	0	0	0	0	bit address	rel. address

Operation:

```
JBC
(PC) ←(PC) + 3
IF (bit) = 1
THEN
    (bit) ←0
    (PC) ←(PC) +rel
```

9.4.26 JC rel

Function: Jump if Carry is set

Description: If the carry flag is set, JC branches to the address indicated; otherwise, it proceeds with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. No flags are affected.

Example: The carry flag is cleared. The following instruction sequence,

```
JC      LABEL1
CPL       C
 JC     LABEL 2
```

sets the carry and causes program execution to continue at the instruction identified by the label LABEL2.

JC rel

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

3 Fast Mode

Encoding:	0	1	0	0	0	0	0	0	rel. address

Operation:

```
JC
(PC) ←(PC) + 2
IF (C) = 1
THEN
    (PC) ←(PC) + rel
```

9.4.27 JMP @A+<base_reg>

Function: Jump indirect

Description: JMP @A+DPTR adds the eight-bit unsigned contents of the Accumulator with the 16-bit data pointer and loads the resulting sum to the program counter. This is the address for subsequent instruction fetches. Sixteen-bit addition is performed (modulo 2^{16}): a carry-out from the low-order eight bits propagates through the higher-order bits. Neither the Accumulator nor the Data Pointer is altered. No flags are affected.

JMP @A+PC adds the eight-bit unsigned contents of the Accumulator to the program counter, which is first incremented by two. This is the address for subsequent instruction fetches. Sixteen-bit addition is performed (modulo 2^{16}): a carry-out from the low-order eight bits propagates through the higher-order bits. The Accumulator is not altered. No flags are affected.

Example: An even number from 0 to 6 is in the Accumulator. The following sequence of instructions branches to one of four AJMP instructions in a jump table starting at JMP_TBL.

```
        MOV     ..... ..... DPTR, # JMP_TBL
        JMP     ..... ..... @A + DPTR
JMP_TBL:AJMP    ..... ..... LABEL0
        AJMP    ..... ..... LABEL1
        AJMP    ..... ..... LABEL2
        AJMP    ..... ..... LABEL3
```

If the Accumulator equals 04H when starting this sequence, execution jumps to label LABEL2. Because AJMP is a 2-byte instruction, the jump instructions start at every other address.

JMP @A+DPTR

Bytes: 1

Cycles: 12 Classic (2 Machine Cycles)

2 Fast Mode

Encoding:	0	1	1	1	0	0	1	1

Operation: (PC) ←(A) + (DPTR)

9.4.27 JMP @A+<base_reg>

JMP @A+PC

Bytes: 2

Cycles: 18 Compatibility (3 Machine Cycles)

3 Fast Mode

Encoding:	1	0	1	0	0	1	0	1	0	1	1	1	0	0	1	1

Operation: (PC) ←(A) + (PC) + 2

9.4.28 JNB bit,rel

Function: Jump if Bit Not set

Description: If the indicated bit is a 0, JNB branches to the indicated address; otherwise, it proceeds with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. *The bit tested is not modified*. No flags are affected.

Example: The data present at input port 1 is 11001010B. The Accumulator holds 56H (01010110B). The following instruction sequence,

```
JNB      P1.3,LABEL1
JNB      ACC.3,LABEL2
```

causes program execution to continue at the instruction at label LABEL2.

JNB bit,rel

Bytes: 3

Cycles: 12 Classic (2 Machine Cycles)

4 Fast Mode

Encoding:	0	0	1	1	0	0	0	0	bit address	rel. address

Operation: (PC) ←(PC) + 3
IF (bit) = 0
THEN (PC) ←(PC) + rel

9.4.29 JNC rel

Function: Jump if Carry not set

Description: If the carry flag is a 0, JNC branches to the address indicated; otherwise, it proceeds with the next instruction. The branch destination is computed by adding the signal relative-displacement in the second instruction byte to the PC, after incrementing the PC twice to point to the next instruction. The carry flag is not modified.

Example: The carry flag is set. The following instruction sequence,

```
JNC      LABEL1
CPL      C
JNC      LABEL2
```

clears the carry and causes program execution to continue at the instruction identified by the label LABEL2.

JNC rel

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

3 Fast Mode

Encoding:	0	1	0	1	0	0	0	0	rel. address

Operation: (PC) ←(PC) + 2
IF (C) = 0
THEN (PC) ←(PC) + rel

9.4.30 JNZ rel

Function: Jump if Accumulator Not Zero

Description: If any bit of the Accumulator is a one, JNZ branches to the indicated address; otherwise, it proceeds with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The Accumulator is not modified. No flags are affected.

Example: The Accumulator originally holds 00H. The following instruction sequence,

```
JNZ     LABEL1
INC       A
JNZ     LABEL2
```

sets the Accumulator to 01H and continues at label LABEL2.

JNZ rel

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

3 Fsat Mode

Encoding:	0	1	1	1	0	0	0	0	rel. address

Operation: JNZ
(PC) ←(PC) + 2
IF (A) ≠ 0
THEN (PC) ←(PC) + rel

9.4.31 JZ rel

Function: Jump if Accumulator Zero

Description: If all bits of the Accumulator are 0, JZ branches to the address indicated; otherwise, it proceeds with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The Accumulator is not modified. No flags are affected.

Example: The Accumulator originally contains 01H. The following instruction sequence,

```
JZ      LABEL1
DEC       A
JZ      LABEL2
```

changes the Accumulator to 00H and causes program execution to continue at the instruction identified by the label LABEL2.

JZ rel

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

3 Fast Mode

Encoding:	0	1	1	0	0	0	0	0	rel. address

Operation: JZ
(PC) ←(PC) + 2
IF (A) = 0
THEN (PC) ←(PC) + rel

9.4.32 LCALL addr16

Function: Long call

Description: LCALL calls a subroutine located at the indicated address. The instruction adds three to the program counter to generate the address of the next instruction and then pushes the 16-bit result onto the stack (low byte first), incrementing the Stack Pointer by two. The high-order and low-order bytes of the PC are then loaded, respectively, with the second and third bytes of the LCALL instruction. Program execution continues with the instruction at this address. The subroutine may therefore begin anywhere in the full 64K byte program memory address space. No flags are affected.

Example: Initially the Stack Pointer equals 07H. The label SUBRTN is assigned to program memory location 1234H. After executing the instruction,

LCALL SUBRTN

at location 0123H, the Stack Pointer will contain 09H, internal RAM locations 08H and 09H will contain 26H and 01H, and the PC will contain 1234H.

LCALL addr16 (Standard Stack)

Bytes: 3

Cycles: 12 Classic (2 Machine Cycles)

4 Fast Mode

Encoding:

0	0	0	1	0	0	1	0	addr15-addr8	addr7-addr0

Operation: LCALL

(PC) ←(PC) + 3
(SP_{7-0}) ←(SP_{7-0}) + 1
((SP_{7-0})) ←(PC_{7-0})
(SP_{7-0}) ←(SP_{7-0}) + 1
((SP_{7-0})) ←(PC_{15-8})
(PC) ←$addr_{15-0}$

LCALL addr16 (Extended Stack)

Bytes: 3

Cycles: 12 Classic (2 Machine Cycles)

6 Fast Mode

Encoding:

0	0	0	1	0	0	1	0	addr15-addr8	addr7-addr0

Operation: LCALL (SP_{15-8} = SPX)

(PC) ←(PC) + 3
(SP_{15-0}) ←(SP_{15-0}) + 1
((SP_{15-0})) ←(PC_{7-0})
(SP_{15-0}) ←(SP_{15-0}) + 1
((SP_{15-0})) ←(PC_{15-8})
(PC) ←$addr_{15-0}$

9.4.33 LJMP addr16

Function: Long Jump

Description: LJMP causes an unconditional branch to the indicated address, by loading the high-order and low-order bytes of the PC (respectively) with the second and third instruction bytes. The destination may therefore be anywhere in the full 64K program memory address space. No flags are affected.

Example: The label JMPADR is assigned to the instruction at program memory location 1234H. The instruction,

LJMP JMPADR

at location 0123H will load the program counter with 1234H.

9.4.33 LJMP addr16

LJMP addr16

Bytes: 3

Cycles: 12 Classic (2 Machine Cycles)

4 Fast Mode

Encoding:	0	0	0	0	0	0	1	0	addr15-addr8	addr7-addr0

Operation: (PC) ←$addr_{15-0}$

9.4.34 LSL M

Function: Shift MAC Accumulator Left Logically

Description: The forty bits in the M register are shifted one bit to the left. Bit 0 is cleared. No flags are affected.

Example: The M register holds the value 0C5B1A29384H. The following instruction,

```
LSL    M
```

leaves the M register holding the value 8B63452708H.

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

2 Fast Mode

Encoding:	A5	0	0	1	0	0	0	1	1

Operation: (M_{n+1}) ←(M_n) n = 0 - 38
(M_0) ← 0

9.4.35 MAC AB

Function: Multiply and Accumulate

Description: MAC AB multiplies the signed 16-bit integers in the register pairs {AX, A} and {BX, B} and adds the 32-bit product to the 40-bit M register. The low-order bytes of the 16-bit operands are stored in A and B, and the high-order bytes in AX and BX respectively. The four operand registers are unaffected by the operation. If the addition of the product to the accumulated sum in M results in a two's complement overflow, the overflow flag is set; otherwise it is not cleared. The carry flag is set if the result is negative and cleared if positive.

Example: Originally the Accumulator holds the value 80 (50H). Register B holds the value 160 (0A0H). The instruction,

```
MAC    AB
```

will give the product 12, 800 (3200H), so B is changed to 32H (00110010B) and the Accumulator is cleared. The overflow flag is set, carry is cleared.

Bytes: 2

Cycles: 30 Classic (5 Machine Cycles)

9 Fast Mode

Encoding:	A5	1	0	1	0	0	1	0	0

Operation: (M_{39-0}) ← (M) + { (AX), (A) } X { (BX), (B) }

9.4.36 MOV <dest-byte>,<src-byte>

Function: Move byte variable

Description: The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected.

This is by far the most flexible operation. Fifteen combinations of source and destination addressing modes are allowed.

Example: Internal RAM location 30H holds 40H. The value of RAM location 40H is 10H. The data present at input port 1 is 11001010B (0CAH).

```
MOV     R0,#30H ..... ..... ;R0 < = 30H
MOV     A,@R0   ..... ..... ;A < = 40H
MOV     R1,A    ..... ..... ;R1 < = 40H
MOV     B,@R1   ..... ..... ;B < = 10H
MOV     @R1,P1  ..... ..... ;RAM (40H) < = 0CAH
MOV     P2,P1   ..... ..... ;P2 #0CAH
```

leaves the value 30H in register 0, 40H in both the Accumulator and register 1, 10H in register B, and 0CAH (11001010B) both in RAM location 40H and output on port 2.

MOV A,R_n

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	1	1	1	0	1	r	r	r

Operation: (A) ←(R_n)

MOV A,direct

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	1	1	1	0	0	1	0	1	direct address

Operation: (A) ←(direct)

MOV A,@R_i

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	1	1	1	0	0	1	1	i

Operation: (A) ←((R_i))

MOV A,#data

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	1	1	1	0	1	0	0	immediate data

Operation: (A) ←#data

9.4.36 MOV <dest-byte>,<src-byte>

MOV R_n,A

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	1	1	1	1	1	r	r	r

Operation: (R_n) ←(A)

MOV R_n,direct

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

2 Fast Mode

Encoding:	1	0	1	0	1	r	r	r	direct addr.

Operation: (R_n) ←(direct)

MOV R_n,#data

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	1	1	1	1	r	r	r	immediate data

Operation: (R_n) ←#data

MOV direct,A

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	1	1	1	1	0	1	0	1	direct address

Operation: (direct) ←(A)

MOV direct,R_n

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

2 Fast Mode

Encoding:	1	0	0	0	1	r	r	r	direct address

Operation: (direct) ←(R_n)

9.4.36 MOV <dest-byte>,<src-byte>

MOV direct,direct

Bytes: 3

Cycles: 12 Classic (2 Machine Cycles)

3 Fast Mode

Encoding:	1	0	0	0	0	1	0	1	dir. addr. (scr)	dir. addr. (dest)

Operation: (direct) ←(direct)

MOV direct,@R_i

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

2 Fast Mode

Encoding:	1	0	0	0	0	1	1	i	direct addr.

Operation: (direct) ←((R_i))

MOV direct,#data

Bytes: 3

Cycles: 12 Classic (2 Machine Cycles)

3 Fast Mode

Encoding:	0	1	1	1	0	1	0	1	direct address	immediate data

Operation: (direct) ←#data

MOV @R_i,A

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	1	1	1	1	0	1	1	i

Operation: ((R_i)) ←(A)

MOV @R_i,direct

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

2 Fast Mode

Encoding:	1	0	1	0	0	1	1	i	direct addr.

Operation: ((R_i)) ←(direct)

9.4.36 MOV <dest-byte>,<src-byte>

MOV @R_i,#data

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	1	1	1	0	1	1	i	immediate data

Operation: ((R_i)) ←#data

9.4.37 MOV DPTR,#data16

Function: Load Data Pointer with a 16-bit constant

Description: MOV DPTR,#data16 loads the Data Pointer with the 16-bit constant indicated. The 16-bit constant is loaded into the second and third bytes of the instruction. The second byte (DPH) is the high-order byte, while the third byte (DPL) holds the lower-order byte. No flags are affected. When multiple data pointers are available the data pointer to load is selected by DPS usually found in the AUXR1 register.

This is the only instruction which moves 16 bits of data at once.

Example: The instruction,

```
MOV  DPTR, #1234H
```

loads the value 1234H into the Data Pointer: DPH holds 12H, and DPL holds 34H.

When two data pointers are available and DPTR0 is selected, the instruction sequence,

```
MOV  DPTR, #1234H
MOV /DPTR, #5678H
```

loads the value 1234H into the first Data Pointer: DPH0 holds 12H and DPL0 holds 34H; and loads the value 5678H into the second Data Pointer: DPH1 hold 56H and DPL1 holds 78H.

MOV DPTR,#data16

Bytes: 3

Cycles: 12 Classic (2 Machine Cycles)

Cycles: 3 Fast Mode

Encoding:	1	0	0	1	0	0	0	0	immed. data15-8	immed. data7-0

Operation: IF DPS = 0
THEN
(DPTR0) ←#$data_{15-0}$
ELSE
(DPTR1) ←#$data_{15-0}$

MOV /DPTR,#data16

Bytes: 4

Cycles: 18 Compatibility (3 Machine Cycles)

Cycles: 4 Fast Mode

Encoding:	A5	90	immed. data15-8	immed. data7-0

Operation: IF DPS = 0
THEN
(DPTR1) ←#$data_{15-0}$
ELSE
(DPTR0) ←#$data_{15-0}$

9.4.38 MOV <dest-bit>,<src-bit>

Function: Move bit data

Description: MOV <dest-bit>,<src-bit> copies the Boolean variable indicated by the second operand into the location specified by the first operand. One of the operands must be the carry flag; the other may be any directly addressable bit. No other register or flag is affected.

Example: The carry flag is originally set. The data present at input Port 3 is 11000101B. The data previously written to output Port 1 is 35H (00110101B).

```
MOV     P1.3,C
MOV     C,P3.3
MOV     P1.2,C
```

leaves the carry cleared and changes Port 1 to 39H (00111001B).

MOV C,bit

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

Cycles: 2 Fast Mode

Encoding:	1	0	1	0	0	0	1	0	bit address

Operation: (C) ←(bit)

MOV bit,C

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

Cycles: 2 Fast Mode

Encoding:	1	0	0	1	0	0	1	0	bit address

Operation: (bit) ←(C)

9.4.39 MOVC A,@A+ <base-reg>

Function: Move Code byte

Description: The MOVC instructions load the Accumulator with a code byte or constant from program memory. The address of the byte fetched is the sum of the original unsigned 8-bit Accumulator contents and the contents of a 16-bit base register, which may be either the Data Pointer or the PC. In the latter case, the PC is incremented to the address of the following instruction before being added with the Accumulator; otherwise the base register is not altered. Sixteen-bit addition is performed so a carry-out from the low-order eight bits may propagate through higher-order bits. No flags are affected.

Example: A value between 0 and 3 is in the Accumulator. The following instructions will translate the value in the Accumulator to one of four values defined by the DB (define byte) directive.

```
REL_PC: INC     ..... A
        MOVC    ..... A,@A+PC
        RET
          DB    ..... 66H
          DB    ..... 77H
          DB    ..... 88H
          DB    ..... 99H
```

If the subroutine is called with the Accumulator equal to 01H, it returns with 77H in the Accumulator. The INC A before the MOVC instruction is needed to “get around” the RET instruction above the table. If several bytes of code separate the MOVC from the table, the corresponding number is added to the Accumulator instead.

9.4.39 MOVC A,@A+ <base-reg>

MOVC A,@A+DPTR

Bytes: 1

Cycles: 12 Classic (2 Machine Cycles)

3 Fast Mode

Encoding:	1	0	0	1	0	0	1	1

Operation:
```
MOVC
IF DPS = 0
THEN
  (A) ←((A) + (DPTR0))
ELSE
  (A) ←((A) + (DPTR1))
```

MOVC A,@A+/DPTR

Bytes: 1

Cycles: 18 Compatibility (3 Machine Cycles)

4 Fast Mode

Encoding:	1	0	0	1	0	0	1	1

Operation:
```
MOVC
IF DPS = 0
THEN
  (A) ←((A) + (DPTR1))
ELSE
  (A) ←((A) + (DPTR0))
```

MOVC A,@A+PC

Bytes: 1

Cycles: 12 Classic (2 Machine Cycles)

3 Fast Mode

Encoding:	1	0	0	0	0	0	1	1

Operation:
```
MOVC
(PC) ←(PC) + 1
(A) ←((A) + (PC))
```

9.4.40 MOVX <dest-byte>,<src-byte>

Function: Move External

Description: The MOVX instructions transfer data between the Accumulator and a byte of external data memory, which is why “X” is appended to MOV. There are two types of instructions, differing in whether they provide an 8-bit or 16-bit indirect address to the external data RAM.

In the first type, the contents of R0 or R1 in the current register bank provide an 8-bit address multiplexed with data on P0. Eight bits are sufficient for external I/O expansion decoding or for a relatively small RAM array. For somewhat larger arrays, any output port pins can be used to output higher-order address bits. These pins are controlled by an output instruction preceding the MOVX.

In the second type of MOVX instruction, the Data Pointer generates a 16-bit address. P2 outputs the high-order eight address bits (the contents of DPH), while P0 multiplexes the low-order eight bits (DPL) with data. The P2 Special Function Register retains its previous contents, while the P2 output buffers emit the contents of DPH. This form of MOVX is faster and more efficient when accessing very large data arrays (up to 64K bytes), since no additional instructions are needed to set up the output ports.

It is possible to use both MOVX types in some situations. A large RAM array with its high-order address lines driven by P2 can be addressed via the Data Pointer, or with code to output high-order address bits to P2, followed by a MOVX instruction using R0 or R1.

9.4.40 MOVX <dest-byte>,<src-byte>

Example: An external 256 byte RAM using multiplexed address/data lines is connected to the 8051 Port 0. Port 3 provides control lines for the external RAM. Ports 1 and 2 are used for normal I/O. Registers 0 and 1 contain 12H and 34H. Location 34H of the external RAM holds the value 56H. The instruction sequence,

```
MOVX  A,@R1
MOVX  @R0,A
```

copies the value 56H into both the Accumulator and external RAM location 12H.

MOVX A,@R_i

Bytes: 1

Cycles: 12 Classic (2 Machine Cycles)

Cycles: 2/4 Fast Mode

Encoding:

1	1	1	0	0	0	1	i

Operation: MOVX
(A) ←((R$_i$))

MOVX A,@DPTR

Bytes: 1

Cycles: 12 Classic (2 Machine Cycles)

2/4 Fast Mode

Encoding:

1	1	1	0	0	0	0	0

Operation: MOVX
IF DPS = 0
THEN
(A) ←((DPTR0))
ELSE
(A) ←((DPTR1)

MOVX A,@/DPTR

Bytes: 1

Cycles: 18 Classic (3 Machine Cycles)

3/5 Fast Mode

Encoding:

1	0	1	0	0	1	0	1

1	1	1	0	0	0	0	0

Operation: MOVX
IF DPS = 0
THEN
(A) ←((DPTR1))
ELSE
(A) ←((DPTR0)

MOVX @R_i,A

Bytes: 1

Cycles: 12 Classic (2 Machine Cycles)

2/4 Fast Mode

Encoding:

1	1	1	1	0	0	1	i

Operation: MOVX
((R$_i$)) ←(A)

9.4.40 MOVX <dest-byte>,<src-byte>

MOVX @DPTR,A

Bytes: 1

Cycles: 12 Classic (2 Machine Cycles)

2/4 Fast Mode

Encoding:

1	1	1	1	0	0	0	0

Operation:
MOVX
IF DPS = 0
THEN
(DPTR0) ←(A)
ELSE
(DPTR1) ←(A)

MOVX @/DPTR,A

Bytes: 1

Cycles: 18 Compatibility (3 Machine Cycles)

3/5 Fast Mode

Encoding:

1	0	1	0	0	1	0	1

1	1	1	1	0	0	0	0

Operation:
MOVX
IF DPS = 0
THEN
(DPTR1) ←(A)
ELSE
(DPTR0) ←(A)

9.4.41 MUL AB

Function: Multiply

Description: MUL AB multiplies the unsigned 8-bit integers in the Accumulator and register B. The low-order byte of the 16-bit product is left in the Accumulator, and the high-order byte in B. If the product is greater than 255 (0FFH), the overflow flag is set; otherwise it is cleared. The carry flag is always cleared.

Example: Originally the Accumulator holds the value 80 (50H). Register B holds the value 160 (0A0H). The instruction,

```
MUL     AB
```

will give the product 12,800 (3200H), so B is changed to 32H (00110010B) and the Accumulator is cleared. The overflow flag is set, carry is cleared.

MUL AB

Bytes: 1

Cycles: 24 Classic (4 Machine Cycles)

2 Fast Mode

Encoding:

1	0	1	0	0	1	0	0

Operation:
MUL
$(A)_{7-0}$ ←(A) X (B)
$(B)_{15-8}$

9.4.42 NOP

Function: No Operation

Description: Execution continues at the following instruction. Other than the PC, no registers or flags are affected.

Example: A low-going output pulse on bit 7 of Port 2 must last exactly 5 cycles. A simple SETB/CLR sequence generates a one-cycle pulse, so four additional cycles must be inserted. This may be done (assuming no interrupts are enabled) with the following instruction sequence,

```
CLR     P2.7
NOP
NOP
NOP
NOP
SETB    P2.7
```

NOP

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	0	0	0	0	0	0	0	0

Operation: NOP
(PC) ←(PC) + 1

9.4.43 ORL <dest-byte> <src-byte>

Function: Logical-OR for byte variables

Description: ORL performs the bitwise logical-OR operation between the indicated variables, storing the results in the destination byte. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

Note: When this instruction is used to modify an output port, the value used as the original port data is read from the output data latch, *not* the input pins.

Example: If the Accumulator holds 0C3H (11000011B) and R0 holds 55H (01010101B) then the following instruction,

```
ORL     A,R0
```

leaves the Accumulator holding the value 0D7H (1101011lB).When the destination is a directly addressed byte, the instruction can set combinations of bits in any RAM location or hardware register. The pattern of bits to be set is determined by a mask byte, which may be either a constant data value in the instruction or a variable computed in the Accumulator at run-time. The instruction,

```
ORL     P1,#00110010B
```

sets bits 5, 4, and 1 of output Port 1.

ORL A,R_n

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	0	1	0	0	1	r	r	r

Operation: ORL
(A) ←(A) ∨ (R_n)

9.4.43 ORL <dest-byte> <src-byte>

ORL A,direct

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	1	0	0	0	1	0	1	direct address

Operation: ORL
(A) ←(A) ∨ (direct)

ORL A,@R_i

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	1	0	0	0	1	1	i

Operation: ORL
(A) ←(A) ∨ ((R_i))

ORL A,#data

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	1	0	0	0	1	0	0	immediate data

Operation: ORL
(A) ←(A) ∨ #data

ORL direct,A

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	1	0	0	0	0	1	0	direct address

Operation: ORL
(direct) ←(direct) ∨ (A)

ORL direct,#data

Bytes: 3

Cycles: 12 Classic (2 Machine Cycles)

3 Fast Mode

Encoding:	0	1	0	0	0	0	1	1	direct addr.	immediate data

Operation: ORL
(direct) ←(direct) ∨ #data

9.4.44 ORL C,<src-bit>

Function: Logical-OR for bit variables

Description: Set the carry flag if the Boolean value is a logical 1; leave the carry in its current state otherwise. A slash (/) preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No other flags are affected.

Example: Set the carry flag if and only if P1.0 = 1, ACC. 7 = 1, or OV = 0:

```
MOV     C,P1.0    ..... ..... ;LOAD CARRY WITH INPUT PIN P10
ORL     C,ACC.7 ..... ..... ;OR CARRY WITH THE ACC. BIT 7
ORL     C,/OV     ..... ..... ;OR CARRY WITH THE INVERSE OF OV.
```

ORLC,bit

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

2 Fast Mode

Encoding:	0	1	1	1	0	0	1	0	bit address

Operation: (C) ←(C) ∨ (bit)

ORLC,/bit

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

2 Fast Mode

Encoding:	1	0	1	0	0	0	0	0	bit address

Operation: (C) ←(C) ∨ ($\overline{bit}$)

9.4.45 POP direct

Function: Pop from stack.

Description: The contents of the internal RAM location addressed by the Stack Pointer is read, and the Stack Pointer is decremented by one. The value read is then transferred to the directly addressed byte indicated. No flags are affected.

Example: The Stack Pointer originally contains the value 32H, and internal RAM locations 30H through 32H contain the values 20H, 23H, and 01H, respectively. The following instruction sequence,

```
POP     DPH
POP     DPL
```

leaves the Stack Pointer equal to the value 30H and sets the Data Pointer to 0123H. At this point, the following instruction,

```
POP     SP
```

leaves the Stack Pointer set to 20H. In this special case, the Stack Pointer was decremented to 2FH before being loaded with the value popped (20H).

POP direct (Standard Stack)

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

2 Fast Mode

Encoding:	1	1	0	1	0	0	0	0	direct address

Operation: POP
(direct) ←((SP_{7-0}))
(SP_{7-0}) ←(SP_{7-0}) - 1

9.4.45 POP direct

POP direct (Extended Stack)

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

3 Fast Mode

Encoding:	1	1	0	1	0	0	0	0	direct address

Operation: POP (SP_{15-8} = SPX)
(direct) ←((SP$_{15-0}$))
(SP$_{15-0}$) ←(SP$_{15-0}$) - 1

9.4.46 PUSH direct

Function: Push onto stack

Description: The Stack Pointer is incremented by one. The contents of the indicated variable is then copied into the internal RAM location addressed by the Stack Pointer. Otherwise no flags are affected.

Example: On entering an interrupt routine, the Stack Pointer contains 09H. The Data Pointer holds the value 0123H. The following instruction sequence,

```
PUSH    DPL
PUSH    DPH
```

leaves the Stack Pointer set to 0BH and stores 23H and 01H in internal RAM locations 0AH and 0BH, respectively.

PUSH direct (Standard Stack)

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

2 Fast Mode

Encoding:	1	1	0	0	0	0	0	0	direct address

Operation: PUSH
(SP$_{7-0}$) ←(SP$_{7-0}$) + 1
((SP$_{7-0}$)) ←(direct)

PUSH direct (Extended Stack)

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

3 Fast Mode

Encoding:	1	1	0	0	0	0	0	0	direct address

Operation: PUSH (SP_{15-8} = SPX)
(SP$_{15-0}$) ←(SP$_{15-0}$) + 1
((SP$_{15-0}$)) ←(direct)

9.4.47 RET

Function: Return from subroutine

Description: RET pops the high- and low-order bytes of the PC successively from the stack, decrementing the Stack Pointer by two. Program execution continues at the resulting address, generally the instruction immediately following an ACALL or LCALL. No flags are affected.

Example: The Stack Pointer originally contains the value 0BH. Internal RAM locations 0AH and 0BH contain the values 23H and 01H, respectively. The following instruction,

```
RET
```

leaves the Stack Pointer equal to the value 09H. Program execution continues at location 0123H.

9.4.47 RET

RET (Standard Stack)

Bytes: 1

Cycles: 12 Classic (2 Machine Cycles)

4 Fast Mode

Encoding:	0	0	1	0	0	0	1	0

Operation: RET
$(PC_{15-8}) \leftarrow((SP_{7-0}))$
$(SP_{7-0}) \leftarrow(SP_{7-0}) - 1$
$(PC_{7-0}) \leftarrow((SP_{7-0}))$
$(SP_{7-0}) \leftarrow(SP_{7-0}) - 1$

RET (Extended Stack)

Bytes: 1

Cycles: 12 Classic (2 Machine Cycles)

5 Fast Mode

Encoding:	0	0	1	0	0	0	1	0

Operation: RET (SP_{15-8} = SPX)
$(PC_{15-8}) \leftarrow((SP_{15-0}))$
$(SP_{15-0}) \leftarrow(SP_{15-0}) - 1$
$(PC_{7-0}) \leftarrow((SP_{15-0}))$
$(SP_{15-0}) \leftarrow(SP_{15-0}) - 1$

9.4.48 RETI

Function: Return from interrupt

Description: For devices with 64KB or less of prgram space, RETI pops the high- and low-order bytes of the PC successively from the stack and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. The Stack Pointer is left decremented by two. No other registers are affected; the PSW is *not* automatically restored to its pre-interrupt status. Program execution continues at the resulting address, which is generally the instruction immediately after the point at which the interrupt request was detected. If a lower- or same-level interrupt was pending when the RETI instruction is executed, that one instruction is executed before the pending interrupt is processed.

For devices with more than 64K bytes of program memory, RETI pops the high-, middle and low-order bytes of the PC successively from the stack and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. The Stack Pointer is left decremented by three.

Example: The Stack Pointer originally contains the value 0BH. An interrupt was detected during the instruction ending at location 0122H. Internal RAM locations 0AH and 0BH contain the values 23H and 01H, respectively. The following instruction,

RETI

leaves the Stack Pointer equal to 09H and returns program execution to location 0123H.

RETI (Standard Stack, 2-byte PC)

Bytes: 1

Cycles: 12 Classic (2 Machine Cycles)

4 Fast Mode

Encoding:	0	0	1	1	0	0	1	0

Operation: $(PC_{15-8}) \leftarrow((SP_{7-0}))$
$(SP_{7-0}) \leftarrow(SP_{7-0}) - 1$
$(PC_{7-0}) \leftarrow((SP_{7-0}))$
$(SP_{7-0}) \leftarrow(SP_{7-0}) - 1$

9.4.48 RETI

RETI (Extended Stack, 2-byte PC)

Bytes: 1

Cycles: 12 Classic (2 Machine Cycles)

5 Fast Mode

Encoding:	0	0	1	1	0	0	1	0

Operation: RETI (SP_{15-8} = SPX)
$(PC_{15-8}) \leftarrow((SP_{15-0}))$
$(SP_{15-0}) \leftarrow(SP_{15-0}) - 1$
$(PC_{7-0}) \leftarrow((SP_{15-0}))$
$(SP_{15-0}) \leftarrow(SP_{15-0}) - 1$

RETI (Standard Stack, 3-byte PC)

Bytes: 1

Cycles: 18 Classic (3 Machine Cycles)

5 Fast Mode

Encoding:	0	0	1	1	0	0	1	0

Operation: RETI
$(PC_{23-16}) \leftarrow((SP_{7-0}))$
$(SP_{7-0}) \leftarrow(SP_{7-0}) - 1$
$(PC_{15-8}) \leftarrow((SP_{7-0}))$
$(SP_{7-0}) \leftarrow(SP_{7-0}) - 1$
$(PC_{7-0}) \leftarrow((SP_{7-0}))$
$(SP_{7-0}) \leftarrow(SP_{7-0}) - 1$

RETI (Extended Stack, 32-byte PC)

Bytes: 1

Cycles: 18 Classic (3 Machine Cycles)

8 Fast Mode

Encoding:	0	0	1	1	0	0	1	0

Operation: RETI (SP_{15-8} = SPX)
$(PC_{23-16}) \leftarrow((SP_{15-0}))$
$(SP_{15-0}) \leftarrow(SP_{15-0}) - 1$
$(PC_{15-8}) \leftarrow((SP_{15-0}))$
$(SP_{15-0}) \leftarrow(SP_{15-0}) - 1$
$(PC_{7-0}) \leftarrow((SP_{15-0}))$
$(SP_{15-0}) \leftarrow(SP_{15-0}) - 1$

9.4.49 RL A

Function: Rotate Accumulator Left

Description: The eight bits in the Accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position. No flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B). The following instruction,

```
RL      A
```

leaves the Accumulator holding the value 8BH (10001011B) with the carry unaffected.

9.4.49 RL A

RL A

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	0	0	1	0	0	0	1	1

Operation: $(A_n + 1) \leftarrow (A_n)$ n = 0 - 6
$(A_0) \leftarrow (A_7)$

9.4.50 RLC A

Function: Rotate Accumulator Left through the Carry flag

Description: The eight bits in the Accumulator and the carry flag are together rotated one bit to the left. Bit 7 moves into the carry flag; the original state of the carry flag moves into the bit 0 position. No other flags are affected.

Example: The Accumulator holds the value 0C5H(11000101B), and the carry is zero. The following instruction,

```
RLC     A
```

leaves the Accumulator holding the value 8BH (10001010B) with the carry set.

RLC A

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast mode

Encoding:	0	0	1	1	0	0	1	1

Operation: $(A_n + 1) \leftarrow (A_n)$ n = 0 - 6
$(A_0) \leftarrow (C)$
$(C) \leftarrow (A_7)$

9.4.51 RR A

Function: Rotate Accumulator Right

Description: The eight bits in the Accumulator are rotated one bit to the right. Bit 0 is rotated into the bit 7 position. No flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B). The following instruction,

```
RR      A
```

leaves the Accumulator holding the value 0E2H (11100010B) with the carry unaffected.

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	0	0	0	0	0	0	1	1

Operation: $(A_n) \leftarrow (A_n + 1)$ n = 0 - 6
$(A_7) \leftarrow (A_0)$

9.4.52 RRC A

Function: Rotate Accumulator Right through Carry flag

Description: The eight bits in the Accumulator and the carry flag are together rotated one bit to the right. Bit 0 moves into the carry flag; the original value of the carry flag moves into the bit 7 position. No other flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B), the carry is zero. The following instruction,

```
RRC     A
```

leaves the Accumulator holding the value 62 (01100010B) with the carry set.

RRC A

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	0	0	0	1	0	0	1	1

Operation: RRC

$(A_n) \leftarrow (A_n + 1)$ n = 0 - 6

$(A_7) \leftarrow (C)$

$(C) \leftarrow (A_0)$

9.4.53 SETB <bit>

Function: Set Bit

Description: SETB sets the indicated bit to one. SETB can operate on the carry flag or any directly addressable bit. No other flags are affected.

Example: The carry flag is cleared. Output Port 1 has been written with the value 34H (00110100B). The following instructions,

```
SETB    C
SETB    P1.0
```

sets the carry flag to 1 and changes the data output on Port 1 to 35H (00110101B).

SETB C

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	1	1	0	1	0	0	1	1

Operation: SETB

$(C) \leftarrow 1$

SETB bit

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	1	1	0	1	0	0	1	0	bit address

Operation: SETB

(bit) $\leftarrow 1$

9.4.54 SJMP rel

Function: Short Jump

Description: Program control branches unconditionally to the address indicated. The branch destination is computed by adding the signed displacement in the second instruction byte to the PC, after incrementing the PC twice. Therefore, the range of destinations allowed is from 128 bytes preceding this instruction 127 bytes following it.

Example: The label RELADR is assigned to an instruction at program memory location 0123H. The following instruction,

```
SJMP    RELADR
```

assembles into location 0100H. After the instruction is executed, the PC contains the value 0123H.

Note: Under the above conditions the instruction following SJMP is at 102H. Therefore, the displacement byte of the instruction is the relative offset (0123H-0102H) = 21H. Put another way, an SJMP with a displacement of 0FEH is a one-instruction infinite loop.

SJMP rel

Bytes: 2

Cycles: 12 Classic (2 Machine Cycles)

2

Encoding:

1	0	0	0	0	0	0	0	rel. address

Operation: SJMP
(PC) ←(PC) + 2
(PC) ←(PC) + rel

9.4.55 SUBB A,<src-byte>

Function: Subtract with borrow

Description: SUBB subtracts the indicated variable and the carry flag together from the Accumulator, leaving the result in the Accumulator. SUBB sets the carry (borrow) flag if a borrow is needed for bit 7 and clears C otherwise. (If C was set *before* executing a SUBB instruction, this indicates that a borrow was needed for the previous step in a multiple-precision subtraction, so the carry is subtracted from the Accumulator along with the source operand.) AC is set if a borrow is needed for bit 3 and cleared otherwise. OV is set if a borrow is needed into bit 6, but not into bit 7, or into bit 7, but not bit 6.

When subtracting signed integers, OV indicates a negative number produced when a negative value is subtracted from a positive value, or a positive result when a positive number is subtracted from a negative number.

The source operand allows four addressing modes: register, direct, register-indirect, or immediate.

Example: The Accumulator holds 0C9H (11001001B), register 2 holds 54H (01010100B), and the carry flag is set. The instruction,

```
SUBB    A,R2
```

will leave the value 74H (01110100B) in the accumulator, with the carry flag and AC cleared but OV set.

Notice that 0C9H minus 54H is 75H. The difference between this and the above result is due to the carry (borrow) flag being set before the operation. If the state of the carry is not known before starting a single or multiple-precision subtraction, it should be explicitly cleared by CLR C instruction.

SUBB A,R_n

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:

1	0	0	1	1	r	r	r

Operation: SUBB
(A) ←(A) - (C) - (R_n)

9.4.55 SUBB A,<src-byte>

SUBB A,direct

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	1	0	0	1	0	1	0	1	direct address

Operation: SUBB
(A) ←(A) - (C) - (direct)

SUBB A,@R_i

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	1	0	0	1	0	1	1	i

Operation: SUBB
(A) ←(A) - (C) - ((R_i))

SUBB A,#data

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	1	0	0	1	0	1	0	0	immediate data

Operation: SUBB
(A) ←(A) - (C) - #data

9.4.56 SWAP A

Function: Swap nibbles within the Accumulator

Description: SWAP A interchanges the low- and high-order nibbles (four-bit fields) of the Accumulator (bits 3 through 0 and bits 7 through 4). The operation can also be thought of as a 4-bit rotate instruction. No flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B). The instruction,
SWAP A
leaves the Accumulator holding the value 5CH (01011100B).

SWAP A

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast mode

Encoding:	1	1	0	0	0	1	0	0

Operation: SWAP
(A_{3-0}) D (A_{7-4})

9.4.57 XCH A,<byte>

Function: Exchange Accumulator with byte variable

Description: XCH loads the Accumulator with the contents of the indicated variable, at the same time writing the original Accumulator contents to the indicated variable. The source/destination operand can use register, direct, or register-indirect addressing.

Example: R0 contains the address 20H. The Accumulator holds the value 3FH (0011111lB). Internal RAM location 20H holds the value 75H (01110101B). The following instruction,

XCH A,@R0

leaves RAM location 20H holding the values 3FH (00111111B) and 75H (01110101B) in the accumulator.

XCH A,R_n

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	1	1	0	0	1	r	r	r

Operation: (A) D ((R_n))

XCH A,direct

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	1	1	0	0	0	1	0	1	direct address

Operation: (A) D (direct)

XCH A,@R_i

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	1	1	0	0	0	1	1	i

Operation: (A) D ((R_i))

9.4.58 XCHD A,@R_i

Function: Exchange Digit

Description: XCHD exchanges the low-order nibble of the Accumulator (bits 3 through 0), generally representing a hexadecimal or BCD digit, with that of the internal RAM location indirectly addressed by the specified register. The high-order nibbles (bits 7-4) of each register are not affected. No flags are affected.

Example: R0 contains the address 20H. The Accumulator holds the value 36H (00110110B). Internal RAM location 20H holds the value 75H (01110101B). The following instruction,

XCHD A,@R0

leaves RAM location 20H holding the value 76H (01110110B) and 35H (00110101B) in the Accumulator.

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	1	1	0	1	0	1	1	i

Operation: XCHD

(A_{3-0}) D ((R_{i3-0}))

9.4.59 XRL <dest-byte>,<src-byte>

Function: Logical Exclusive-OR for byte variables

Description: XRL performs the bitwise logical Exclusive-OR operation between the indicated variables, storing the results in the destination. No flags are affected.

The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

Note: When this instruction is used to modify an output port, the value used as the original port data is read from the output data latch, *not* the input pins.

Example: If the Accumulator holds 0C3H (1100001lB) and register 0 holds 0AAH (10101010B) then the instruction,

```
XRL     A,R0
```

leaves the Accumulator holding the value 69H (01101001B).

When the destination is a directly addressed byte, this instruction can complement combinations of bits in any RAM location or hardware register. The pattern of bits to be complemented is then determined by a mask byte, either a constant contained in the instruction or a variable computed in the Accumulator at run-time. The following instruction,

```
XRL     P1,#00110001B
```

complements bits 5, 4, and 0 of output Port 1.

XRL A,R_n

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

1 Fast Mode

Encoding:	0	1	1	0	1	r	r	r

Operation: (A) ←(A) ⊻ (R_n)

XRL A,direct

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	1	1	0	0	1	0	1	direct address

Operation: (A) ←(A) ⊻ (direct)

XRL A,@R_i

Bytes: 1

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	1	1	0	0	1	1	i

Operation: (A) ←(A) ⊻ ((R_i))

XRL A,#data

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	1	1	0	0	1	0	0	immediate data

Operation: (A) ←(A) ⊻ #data

9.4.59 XRL <dest-byte>,<src-byte>

XRL direct,A

Bytes: 2

Cycles: 6 Classic (1 Machine Cycle)

2 Fast Mode

Encoding:	0	1	1	0	0	0	1	0	direct address

Operation: (direct) ←(direct) ⊻ (A)

XRL direct,#data

Bytes: 3

Cycles: 12 Classic (2 Machine Cycles)

3 Fast Mode

Encoding:	0	1	1	0	0	0	1	1	direct address	immediate data

Operation: (direct) ←(direct) ⊻ #data

Section 10

Appendix A - Error Messages

10.1 Assembly Errors

Assembly errors apply to the consistency of the assembly language program in syntax and semantics. If one of these errors is detected, it is flagged in the list file, and program execution continues. When assembly is finished, Atmel® C51ASM terminates with exit code 1:.

10.1.1 ERROR #1

Summary:

ERROR #1: Unrecognized input

Description:

A statement contains characters that are not allowed in assembly language.

10.1.2 ERROR #2

Summary:

ERROR #2: Undefined symbol

Description:

A symbol is referenced, but has never been defined.

10.1.3 ERROR #3

Summary:

ERROR #3: Duplicate symbol

Description:

Attempt to redefine a previously defined symbol.

Summary:

ERROR #3: Duplicate label

Description:

Attempt to define a previously defined label symbol.

10.1.4 ERROR #4

Summary:

ERROR #4: Illegal digit for radix

Description:

An illegal digit for the specified radix was found in a numeric constant.

10.1.5 ERROR #5

Summary:

ERROR #5: Expression greater than 8-bits

Description:

An expression expected to evaluate as an 8-bit address is larger than 255.

Summary:

ERROR #5: DATA address out of range

Description:

An expression expected to evaluate as an 8-bit address is larger than 255.

10.1.6 ERROR #6

Summary:

ERROR #6: Missing END directive

Description:

The program ends without an END statement.

10.1.7 ERROR #8

Summary:

ERROR #8: Illegal assembly line

Description:

A statement contains a syntax element that is not allowed in this context.

10.1.8 ERROR #9

Summary:

ERROR #9: command past END statement

Description:

The END statement is followed by further non-comment assembler statements.

10.1.9 ERROR #12

Summary:

ERROR #12: Unbalanced parentheses (missing open)

Description:

An expression contains more closing than opening parentheses.

Summary:

ERROR #12: Unbalanced parentheses (missing closed)

Description:

An expression contains more opening than closing parentheses.

10.1.10 ERROR #15

Summary:

ERROR #15: Illegal dptr instruction

Description:

DPTR instruction aliases are only valid with the --dptr option.

10.1.11 ERROR #16

Summary:

ERROR #16: Divide by zero

Description:

During evaluation of the expression, the assembler has to divide by zero.

10.1.12 ERROR #17

Summary:

ERROR #17: Invalid base address

Description:

A DATA address that is not bit-addressable has been used on the left side of a '.' operator.

Summary:

ERROR #17: Invalid bit number

Description:

A number greater than 7 has been used on the right side of a '.' operator.

10.1.13 ERROR #18

Summary:

ERROR #18: Relative offset exceeds -128 / +127

Description:

The target address of a relative jump instruction cannot be reached within +127 or -128 bytes.

10.1.14 ERROR #19

Summary:

ERROR #19: Address outside current 2K page

Description:

The target address of an AJMP or ACALL instruction is not located within the same 2048 byte page as the next instruction.

10.1.15 ERROR #20

Summary:

ERROR #20: Illegal operand

Description:

One of the operands specified is not a legal operand for the instruction.

Summary:

ERROR #20: Illegal 2nd operand

Description:

The second operand is not a legal operand for the instruction.

Summary:

ERROR #20: Illegal 3rd operand

Description:

The third operand is not a legal operand for the instruction.

Summary:

ERROR #20: Illegal operand, expecting bit

Description:

Bit operation with non-bit operand.

10.1.16 ERROR #21

Summary:

ERROR #21: Illegal indirect register

Description:

Expected R0 or R1. An indirect addressing symbol (@) is used with something other than R0 or R1.

10.1.17 ERROR #23

Summary:

ERROR #23: Unsupported directive

Description:

The assembler does not support the relocatable directives: PUBLIC, EXTRN, SEGMENT and RSEG.

Summary:

ERROR #23: EDATA Segment not implemented

Description:

The selected device does not implement EDATA.

Summary:

ERROR #23: FDATA Segment not implemented

Description:

The selected device does not implement FDATA.

10.1.18 ERROR #24

Summary:

ERROR #24: Attempting to EQU a previously SET symbol

Description:

A symbol defined with the SET directive cannot later be redefined using the EQU directive.

10.1.19 ERROR #25

Summary:

ERROR #25: Attempt to SET an EQU symbol

Description:

A symbol defined with the EQU directive cannot later be redefined using the SET directive.

10.1.20 ERROR #27

Summary:

ERROR #27: Target address is undefined

Description:

The expression for the target address contains a symbol that is not defined.

Summary:

ERROR #27: Target address has wrong type

Description:

The expression for the target address is not typeless and does not have type CODE.

10.1.21 ERROR #28

Summary:

ERROR #28 Address below segment base

Description:

Attempt to set the location counter of the current segment below the segment base address.

Summary:

ERROR #28 FDATA segment below base

Description:

An FSEG AT statement attempts to define an FDATA segment that starts below the lowest logical address for the selected device. Normally this occurs when an FSEG is set at zero, but the selected device has FDATA starting at some higher address.

Summary:

ERROR #28 BIT segment above limit
ERROR #28 CODE segment above limit
ERROR #28 DATA segment above limit
ERROR #28 EDATA segment above limit

ERROR #28 FDATA segment above limit
ERROR #28 IDATA segment above limit

Description:

Attempt to set the location counter of the current segment above the highest logical address for this segment in the selected device.

10.1.22 ERROR #30

Summary:

ERROR #30: DB not allowed in this segment!
ERROR #30: DW not allowed in this segment!

Description:

The DB and DW directives are only allowed in CODE or FDATA segments.

Summary:

ERROR #30: DBIT only allowed in BSEG!

Description:

The DBIT directive is only allowed in a BIT segment.

Summary:

ERROR #30: DS not allowed in BSEG!

Description:

The DS directive is not allowed in a BIT segment. Use DBIT.

Summary:

ERROR #30: mnemonic only allowed in CSEG

Description:

Instruction mnemonics are only allowed in a CODE segment.

10.1.23 ERROR #31

Summary:

ERROR #31: Character strings not allowed

Description:

A string with more than two characters is used in a DW statement. Use DB to generate string values.

10.1.24 ERROR #33

Summary:

ERROR #33: String not terminated

Description:

A character string is not properly terminated with a quote.

10.1.25 ERROR #34

Summary:

ERROR #34: USING must be between 0-3

Description:

The register bank number specified with the USING directive exceeds 3.

10.1.26 ERROR #35

Summary:

ERROR #35: $RESTORE must be preceded by $SAVE

Description:

A $RESTORE control must be preceded by a $SAVE control.

10.1.27 ERROR #37

Summary:

ERROR #37: Illegal control statement

Description:

A statement is starting with an unknown keyword beginning with a $.

Summary:

ERROR #37: Control preceded by non-control lines

Description:

A primary control occurs after statements that are not assembler controls.

Summary:

ERROR #37: Unexpected input in control

Description:

The assembler was unable to process part of a control statement.

10.1.28 ERROR #40

Summary:

ERROR #40: Synchronization error

Description:

A label has different values defined on pass 1 and pass 2.

Summary:

ERROR #40: Address overlaps

Description:

Machine code has already been generated for the locations the current statement is targeting.

10.1.29 ERROR #41

Summary:

ERROR #41: Instruction requires extended memory mode

Description:

On of the extended memory instructions: EJMP, ECALL or ERET, was found, but the assembler is not in extended memory mode.

10.1.30 ERROR #43

Summary:

ERROR #43: No matching IF
ERROR #43: ELSE with no IF
ERROR #43: ELSEIF with no IF

Description:

An ELSEIF, ELSE or ENDIF conditional assembly directive occurs without a preceding IF directive.

Summary:

ERROR #43: Wrong conditional type

Description:

All IF-ELSE-ENDIF directives in a conditional assembly block must be of the same type.

10.1.31 ERROR #44

Summary:

ERROR #44: Missing ENDIF

Description:

A conditional assembly block as found without a terminating ENDIF directive.

10.1.32 ERROR #45

Summary:

ERROR #45: Illegal or missing macro name

Description:

The macro name is missing from a macro definition or is already defined as another symbol.

10.1.33 ERROR #46

Summary:

ERROR #46: ENDM without MACRO

Description:

An ENDM directive occurs without a preceding macro definition.

10.1.34 ERROR #47

Summary:

ERROR #47: LOCAL not allowed in MACRO body

Description:

Macro body lines occur before a LOCAL statement in a macro definition. Move the LOCAL statement before any body lines.

Summary:

ERROR #47: LOCAL instruction outside macro!

Description:

A LOCAL statement is illegal outside of a macro definition.

10.1.35 ERROR #48

Summary:

ERROR #48: Illegal parameter specification

Description:

The parameter names of a macro are not all different.

10.1.36 ERROR #50

Summary:

ERROR #50: Expected one of '(', '<; or 'â?œ'

Description:

Include file string is missing or improperly delimited.

Summary:

ERROR #50: Include string not terminated, expected ')"
ERROR #50: Include string not terminated, expected '>'
ERROR #50: Include string not terminated, expected '"'

Description:

Include file string is not terminated.

10.2 Runtime Errors

Runtime errors are operational errors, or I/O errors. If one of these errors is detected, it is flagged on the console, and Atmel C51ASM aborts with exit code 2.

10.2.1 I/O ERROR

I/O ERROR: No such file or directory

Source or include file not found. (Linux)

I/O ERROR: Is a directory

Object or listing file is actually a directory. (Linux)

I/O ERROR: Permission denied

Source or include file does not have the correct privileges. (Linux)

I/O ERROR: no source file specified

The source file is missing from the command line.

I/O ERROR: Unknown option

The specified command line option is not recognized

I/O ERROR: No argument supplied with option

The specified command line option requires an argument.

I/O ERROR: missing argument for option

The specified command line option requires an argument to be attached to the option.

I/O ERROR: wrong file type

The specified file format is not supported.

10.2.2 FATAL ERROR

FATAL ERROR: Could not open input file

Source or include file cannot be opened for input.

FATAL ERROR: Could not open file for writing

Object or listing file cannot be opened for output.

FATAL ERROR: Unable to allocate memory

No free memory is available.

Section 11

Appendix B - List File Format

The listing file displays the results of the translation from assembly source to machine code. This appendix describes the format of the listing file generated by the assembler.

By default the Atmel® C51ASM assembler does not generate a listing file. To produce a list file output, include --listfile on the command line or place a $PRINTcontrol in the source file.

11.1 List File Headings

Normally, every page of the listing has a page header on the first line. The header identifies the assembler and version, and lists the title, date and page number. If a title was not specified with the $TITLE control, the assembler copyright information is substituted. When a listing page reaches the maximum number of lines per page, a form feed character is output, followed by another page header. The page header and form feeds can be suppressed with the --nopaging option or the $NOPAGING control. Use the $PAGELENGTH control to adjust the number of lines per page. The page header (and all other lines) will be truncated to fit the page width set by the $PAGEWIDTH control.

```
C51ASM V1.0     Copyright (c) 2010 Atmel Corp.     2010/09/01 PAGE 1
```

The first page of the listing contains the file header in addition to the page header. The file header identifies the assembler and version, lists all input and output files and marks the columns for the line headings. A typical file header is looking as shown below:

```
        8051 Macro Assembler   C 5 1 A S M   V 1.0
        ========================================



        Source File:    demo.asm
        Object File(s): demo.hex
        List File:      demo.lst



Line  I  Addr Code            Source
```

11.2 Source Listing

The formatted source listing makes up the main body of the listing file. The format for each line in the listing file depends on the type of source line that appears on it. Standard source lines contain five fields, corresponding to the five column headings printed with each page header, as described below:

- **Line**

Contains the global line number, counting the number of lines processed from the top of the program. This includes lines from include files and macro expansions. Therefore, the global line numbers may not necessarily correspond to actual assembly source line numbers within the particular source files. The line number field is terminated by a colon character (:) for source lines from the main source and all include files, and with a plus character (+) for macro expansion lines.

- **I**
  Flags the level of include file or macro nesting. This column is always empty in the body of the main source file. If a file is included, the number of open include files id displayed in the "I" column. When a macro is expanded, the number of open macro calls appended to the '+' following the line number.
- **Addr**
  Shows the hexadecimal start address of the listed line in the currently active segment, if it generates code or reserves space. The field is left blank if a line cannot be assigned to a particular segment.
- **Code**
  Shows the actual machine code produced by the assembly line. The code is listed in hexadecimal byte quantities starting from the left margin of the "Code" column. In the case of DB or DW directives that generate code longer than four bytes, the source line is followed by additional lines containaing just the "Addr" and "Code" fields until the whole generated code of the line is listed.

  If the assembled line contains an expression, but does not cosume any address space, the assembler lists the evaluated hexadecimal result of the expression at the right margin of the "Code" column. The segment type of the expression is flagged with one single character at the left margin of the "Code" column as shown in Table 11-1.

Table 11-1. List File Expression Types

Flag	Type
C	CODE
D	DATA
I	IDATA
E	EDATA
F	FDATA
X	XDATA
B	BIT
N	typeless number
R	register

- **Source**
  Lists the original assembly source line as it appears in the source file. However, the line may be truncated to fit the specified page width.

Example:

```
Line  I  Addr Code            Source

   1:                         ; Uart Echo Sample Program
   2:                         ;==========================
   3:
   4:                         $TITLE(UART ECHO SAMPLE PROGRAM)
   5:                         $MOD52 PW(64) NOBUILTIN
   6:
```

```
  7:          B          00     RDY BIT 20h.0
  8:          B          01     BUF_FULL BIT 20h.1
  9:
 10:
 11:          N        0008     ISEG AT 8
 12:       08 N          08     STACK: DS 8
 13:
 14:          N        0000     CSEG AT 0
 15:
 16:     0000 02 00 3C          LJMP START
 17:
 18:          N        0023     ORG 23h
 19: 0023 10 99 0A              SINTR:  JBC TI, TX_INT
 20: 0026 C2 98                 RX_INT: CLR RI
 21: 0028 E5 99                         MOV A, SBUF
 22: 002A 20 00 08                      JB RDY, SEND
 23: 002D D2 01                         SETB BUF_FULL
 24: 002F 32                            RETI
 25:
 26: 0030 D2 00                 TX_INT: SETB RDY
 27: 0032 30 01 06                      JNB BUF_FULL, EXIT
 28: 0035 F5 99                 SEND:   MOV SBUF, A
 29: 0037 C2 00                         CLR RDY
 30: 0039 C2 01                         CLR BUF_FULL
 31: 003B 32                    EXIT:   RETI
```

Errors

When an error is detected on a source line, its position is marked with a caret character (^) and a comprehensive error message is inserted. Note that the marked location of the error is only approximate.

```
  29: 0037 C2 00                        CLR RDY
  30: 0039 C2 00                        CLR BUF_FUL
                                            ^
ERROR #2:            @@@@@ Undefined symbol @@@@@

  31: 003B 32                   EXIT: RETI
```

The error diagnosis at the end of program lists the segment usage, which register banks were used, and the total number of warnings and errors detected throughout the assembly:

```
Segment usage:
   Code      :   687 bytes
   Data      :    35 bytes
   Idata     :    16 bytes
   Edata     :    0 bytes
   Fdata     :    0 bytes
   Xdata     :    5 bytes
   Bit       :    4 bits

   Register banks used: 0, 1, 3

   Warnings: 0
   Errors:   7
```

A register bank counts as "used", if the program had switched to that bank with a USING instruction, or one of the special assembler symbols AR0 ... AR7 has been used, while the bank was active. The message

```
register banks used: ---
```

means, that no bank has been used explicitly, and that the program code may, but need not, be register bank independent.

11.3 Symbol Table

The symbol table or cross-reference listing is the last section of the listing file. By default, a symbol table is generated. The symbol table is a list of all the symbols defined in a program, including predefined symbols, with their type, value and first line of definition as described below:

- The **SYMBOL** field lists the names of the symbols in alphabetical order. The name is trucated to a maximum of 31 characters.
- The **TYPE** field specifies the segment type of the symbol, or REGISTER or MACRO if the symbol is a register alias or macro name.
- The **VALUE** field shows the hexadecimal value of the symbol or the symbolic register for aliases.
- The **LINE** field contains the global line number where the symbol was first defined. The LINE field is blank for predefined symbols.
- The symbol table listing can be suppressed with the $NOSYMBOLS control. A typical symbol table listing is shown below:

```
      L I S T   O F   S Y M B O L S
      =============================


SYMBOL                          TYPE     VALUE     LINE
------------------------------------------------------
BUF_FULL                        BIT        01        8
EXIT                            CODE     003B       31
LP51                            UNDEF                0
RDY                             BIT        00        7
RX_INT                          CODE     0026       20
SEND                            CODE     0035       28
SINTR                           CODE     0023       19
STACK                           IDATA      08       12
START                           CODE     003C       33
TX_INT                          CODE     0030       26
USE_AUTOBAUD                    UNDEF                0
```

If the $XREF control is specified, a cross-reference listing is generated instead of a symbol table. It lists all the symbols of the program in alphabetical order, with their symbol name, all definitions including definition lines, segment types, and numerical values. Furthermore, all symbol references are listed as well. The SYMBOL column contains the symbol name, while the columns TYPE, VALUE, and DEFINED may contain the segment types, numerical values, and definition lines of one, more, or no symbol defintions.

- The **SYMBOL** field lists the names of the symbols in alphabetical order. The name is trucated to a maximum of 31 characters.
- The **TYPE** field specifies the segment type of the symbol, or REGISTER or MACRO if the symbol is a

register alias or macro name.

- The **VALUE** field shows the hexadecimal value of the symbol or the symbolic register for aliases. If the symbol is defined multiple times (e.g a SET symbol) then each value is listed on separate lines.
- The **DEFINED** field contains the global line number where the symbol was defined. The LINE field is blank for predefined symbols. If the symbol is defined multiple times (e.g a SET symbol) then each vdefinition line numbers is listed on a separate line.
- The **REFERENCED** field contains a list of each global line number where the symbol was referenced. The list of references may appear on multiple lines.

The cross-reference listing can be suppressed with the $NOXREF control. The corresponding cross-reference listing for the symbol table above is as follows:

```
C R O S S - R E F E R E N C E - L I S T I N G
=============================================


SYMBOL                        TYPE     VALUE  DEFINED  REFERENCED
-------------------------------------------------------------------
BUF_FULL                      BIT        01        8   23   27
                                                     30   50
EXIT                          CODE     003B       31   27
LP51                          undef                0   39
RDY                           BIT        00        7   22   26
                                                     29   49
RX_INT                        CODE     0026       20
SEND                          CODE     0035       28   22
SINTR                         CODE     0023       19
STACK                         IDATA      08       12
START                         CODE     003C       33   16
TX_INT                        CODE     0030       26   19
USE_AUTOBAUD                  undef                0   34   56
```

Section 12

Appendix C - Predefined Symbols

The folling symbols are predefined by the $MOD51 and $MOD52 assembler controls. Note that the $MOD52 control also includes the $MOD51 symbols. To prevent these symbols from appearing in the symbol table, use the $NOBUILTIN control. $MOD51 is on by default. To disable the predefined symbols, use the $NOMOD51 control.

Table 12-1. $MOD51 Definitions

DATA Addresses				BIT Addresses			
Symbol	**Value**	**Symbol**	**Value**	**Symbol**	**Value**	**Symbol**	**Value**
P0	080H	P1	090H	IT0	088H	EA	0AFH
SP	081H	SCON	098H	IE0	089H	RXD	0B0H
DPL	082H	SBUF	099H	IT1	08AH	TXD	0B1H
DPH	083H	P2	0A0H	IE1	08BH	INT0	0B2H
PCON	087H	IE	0A8H	TR0	08CH	INT1	0B3H
TCON	088H	P3	0B0H	TF0	08DH	T0	0B4H
TMOD	089H	IP	0B8H	TR1	08EH	T1	0B5H
TL0	08AH	PSW	0D0H	TF1	08FH	WR	0B6H
TL1	08BH	ACC	0E0H	RI	098H	RD	0B7H
TH0	08CH	B	0F0H	TI	099H	PX0	0B8H
TH1	08DH			RB8	09AH	PT0	0B9H
Plain Numbers				TB8	09BH	PX1	0BAH
Symbol	**Value**	**Symbol**	**Value**	REN	09CH	PT1	0BBH
??C51ASM	8051H	??VERSION	0100H	SM2	09DH	PS	0BCH
				SM1	09EH	P	0D0H
				SM0	09FH	OV	0D2H
				EX0	0A8H	RS0	0D3H
				ET0	0A9H	RS1	0D4H
				EX1	0AAH	F0	0D5H
				ET1	0ABH	AC	0D6H
				ES	0ACH	CY	0D7H

The ??C51ASM and ??VERSION symbols are always active and cannot be switched off by the $NOMOD51.

Table 12-2. $MOD52 Definitions

DATA Addresses				BIT Addresses			
Symbol	**Value**	**Symbol**	**Value**	**Symbol**	**Value**	**Symbol**	**Value**
T2CON	0C8H	T2MOD	0C9H	T2	090H	T2EX	091H
RCAP2L	0CAH	RCAP2H	0CBH	ET2	0ADH	PT2	0BDH
TL2	0CCH	TH2	0CDH	CPRL2	0C8H	CT2	0C9H
				TR2	0CAH	EXEN2	0CBH
				TCLK	0CCH	RCLK	0CDH
				EXF2	0CEH	TF2	0CFH

Section 13

Appendix D - Reserved Keywords

The Atmel® C51ASM assembler uses predefined symbols or reserved keywords that may not be redefined in your program. Reserved names include instruction mnemonics, directives, operators, and register names. The following tables list the reserved symbol names:

Table 13-1. Special Assembler Symbols

$	A	AB	AR0	AR1	AR2
AR3	AR4	AR5	AR6	AR7	C
DPTR	PC	R0	R1	R2	R3
R4	R5	R6	R7		

Table 13-2. Instruction Mnemonics

ACALL	ADD	ADDC	AJMP	ANL	CALL
CJNE	CLR	CPL	DA	DEC	DIV
DJNZ	INC	JB	JBC	JC	JMP
JNB	JNC	JNZ	JZ	LCALL	LJMP
MOV	MOVC	MOVX	MUL	NOP	ORL
POP	PUSH	RET	RETI	RL	RLC
RR	RRC	SETB	SJMP	SUBB	SWAP
XCH	XCHD	XRL			

Table 13-3. Assembler Directives

AT	BIT	BSEG	CODE	CSEG	DATA
DB	DBIT	DS	DSEG	DW	EDATA
ELSE	ELSEIF	ELSEIFB	ELSEIFDEF	ELSEIFN	ELSEIFNB
ELSEIFNDEF	END	ENDIF	ENDM	ENDMACRO	EQU
ESEG	EXITM	EXTRN	FDATA	FSEG	IDATA
IF	IFB	IFDEF	IFN	IFNB	IFNDEF
ISEG	LOCAL	MACRO	NAME	ORG	PUBLIC
REPT	RSEG	SEGMENT	SET	SFR	USING
XDATA	XSEG				

Table 13-4. Extended Instruction Mnemonics

ASR	BREAK	ECALL	EJMP	ERET	LSL
MAC					

Table 13-5. Operators

AND	EQ	GE	GT	HIGH	LE
LOW	LT	MOD	NE	NOT	OR
SHL	SHR	XOR			

Table 13-6. Assembler Controls

$COND	$CONDONLY	$DA	$DATE	$DB	$DEBUG
$DEVICE	$EJ	$EJECT	$ERROR	$GE	$GEN
$GENONLY	$GO	$IC	$INCDIR	$INCLUDE	$LI
$LIST	$MACRO	$MAPFSEG	$MESSAGE	$MO	$MOD51
$MR	$NOBUILTIN	$NOCOND	$NODB	$NODEBUG	$NOGE
$NOGEN	$NOLI	$NOLIST	$NOMACRO	$NOMO	$NOMOD51
$NOMR	$NOOBJECT	$NOOJ	$NOPAGING	$NOPI	$NOPR
$NOPRINT	$NOSB	$NOSYMBOLS	$NOTABS	$NOXR	$NOXREF
$OBJECT	$OJ	$PAGELENGTH	$PAGEWIDTH	$PAGING	$PI
$PL	$PR	$PRINT	$PW	$RESTORE	$RS
$SA	$SAVE	$SB	$SYMBOLS	$TITLE	$TT
$WARNING	$XR	$XREF			

Section 14

Appendix E - Object File Formats

14.1 Intel-HEX Format

The Intel® HEX object file format is supported by many assemblers, compilers, utilities and device programmers to specify executable byte code for embedded processors. An Intel HEX file is a 7-bit ASCII text file that contains a sequence of data records, address records and an end record. Every record is a line of text that starts with a colon and ends with CR and LF. Each record contains an 8-bit checksum to identify corrupted data.

Each line of an Intel HEX file consists of six fields:

1. **Start code**: one character, an ASCII colon ':'.
2. **Byte count**: two hex digits representing the number of bytes (hex digit pairs) in the data field, from 0–255. C51ASM outputs 16 bytes or fewer per record.
3. **Address**: four hex digits representing the 16-bit starting memory address for the data within a 64 KB block. The 64 KB limit is exceeded by specifying higher address bits via additional record types. The address is big endian (e.g. higher order byte first).
4. **Record type**: two hex digits, 00 to 05, defining the type of the data field.
5. **Data**: a sequence of *N* bytes of the data themselves (2*N* hex digits).
6. **Checksum**: two hex digits representing the two's complement modulo 256 sum of the values in all fields except fields 1 and 6 (Start code ":" and Checksum). It is calculated by adding together the hex-encoded bytes (hex digit pairs) as an 8-bit sum (leaving only the LSB of the result) and making a 2's complement. If the checksum is correctly calculated, adding all the bytes (the byte count, both bytes in address, the record type, each data byte and the checksum) together will always result in a value wherein the least significant byte is zero (0x00).

   For example, on :0300300002337A1E, calculate 03 + 00 + 30 + 00 + 02 + 33 + 7A = E2, and 2's complement is 1E

Table 14-1. Intel-Hex Record Types

Type	Name	Description
00	Data Record	Contains data and 16-bit offset address. The format is as described above.
01	End Of File Record	The file termination record. The data field is empty. Has to be the last line of the file, and only one per file permitted. The Intel specification allows the End Of File record to contain a start address for the program being loaded, e.g. :00AB2F0125 would make a jump to address AB2F. For C51ASM the address field is always zero, e.g. ':00000001FF'.
02	Extended Segment Address Record	One megabyte (20-bit) segment-base address. Used when the object file exceeds 64 KB. The address specified by the 02 record is multiplied by 16 (shifted 4 bits left) and added to the subsequent 00 record addresses. The address field of this record is 0000 and the byte count is 02 (the segment is 16-bit, big endian). In the Intel specification the least significant hex digit of the segment address is always 0. For C51ASM the segment is fixed on 64 KB boundaries making the three least significant hex digits always 0, for example ':0200000230009C' starts the segment at 0x30000.
03	Start Segment Address Record	Specifies the initial contents of the CS:IP registers for 80x86 processors, but is not used in C51ASM. The address field is 0000, the byte count is 04, the first two bytes are the CS value, and the latter two are the IP value.
04	Extended Linear Address Record	Specifies the upper address word for full 32-bit (4GB) addressing. The address field is 0000, the byte count is 02 and the two data bytes represent the upper 16 bits of the 32 bit address. C51ASM supports only up to 24-bit (16MB) addressing, so the higher order byte is always zero.
05	Start Linear Address Record	Specfies the value of the EIP register on 80386 and higher CPUs. The address field is 0000, the byte count is 04 and the 4 data bytes represent the 32-bit value loaded into the EIP register. C51ASM does not use this record type.

Example:

```
:10010000214601360121470136007EFE09D2190140
:100110002146017EB7C20001FF5F16002148011988
:10012000194E79234623965778239EDA3F01B2CAA7
:100130003F0156702B5E712B722B732146013421C7
:00000001FF
```

14.2 Motorola SREC Format

Motorola S-records (SREC) are an alternative to the Intel HEX files that are supported by some device programmers, especially for devices with memory above 64KB. An SREC format object file consists of a series of ASCII records. All hexadecimal numbers are big endian. The records have the following structure.

Each line of an SREC file consists of six fields:

1. **Start code**: one character, an ASCII 'S'.
2. **Record type**: one ASCII digit, 0–9, defining the type of the data field.
3. **Byte count**: two hex digits indicating the number of bytes (hex digit pairs) that follow in the rest of the record (in the address, data and checksum fields).

4. **Address**: four, six, or eight hex digits as determined by the record type for the memory location of the first data byte in the data field.
5. **Data**: a sequence of *N* bytes of data (2*N* hex digits).
6. **Checksum**: two hex digits representing the one's complement of the modulo 256 sum of the values represented by the hex digit pairs for the byte count, address and data fields.

There are eight record types:

Table 14-2. Motorola SREC Record Types

Type	Name	Address Bytes	Data Allowed	Description
S0	Block header	2	Yes	First record in a file. Contains vendor specific data. Address field is always zero.
S1	Data sequence	2	Yes	Contains data and 16-bit offset address for a 64KB block.
S2	Data sequence	3	Yes	Contains data and 24-bit offset address for a 16MB block.
S3	Data sequence	4	Yes	Contains data and 32-bit offset address for a 4GB block.
S5	Record count	2	No	The address field contains the number of S1, S2 or S3 record is the file.
S7	End of block	4	No	Terminates a file containing S3 records. The address field is the 32-bit starting address.
S8	End of block	3	No	Terminates a file containing S2 records. The address field is the 24-bit starting address.
S9	End of block	2	No	Terminates a file containing S1 records. The address field is the 16-bit starting address.

The **S0** record data sequence contains vendor specific data rather than program data. The data includes thee fixed length fields and one variable length field as described below. The offset column refers to the starting position within the data sequence.

Table 14-3. S0 Vendor Fields

Field	Offset	Size	Desciption
Module	0	20	Ten characters representing the ASCII values of the module name
Version	20	2	A single character representing the version number in ASCII
Revision	22	2	A single character representing the revision number in ASCII
Comment	24	0-36	0-18 characters repsenting the ASCII values of a text comment

Example:

```
S00F000068656C6C6F202020202000003C
S11F00007C0802A6900100049421FFF07C6C1B787C8C23783C6000003863000026
S11F001C4BFFFFE5398000007D83637880010014382100107C0803A64E800020E9
S111003848656C6C6F20776F726C642E0A0042
S5030003F9
S9030000FC
```

14.3 Intel Object Module Format

The Atmel® C51ASM assembler can generate object files in the Intel Object Module Format for 8051s (OMF-51). Only the Absolute OMF subset is supported as C51ASM is not a relocatable assembler. OMF-51 objects can optionally contain debugging information including the symbol table and source line number mappings.

The original OMF-51 specification is available on the Internet:

- http://plit.de/asem-51/omf51eps.htm (online)
- http://www.keil.com/download/files/omf51.zip (download zipped PDF)

14.3.1 Extensions

C51ASM implements the following extensions to the OMF-51 specification:

- The **SYM INFO** field of Debug Item Records (type 12H) has extended **USAGE TYPE** values for the expanded C51ASM segment types. A value of 6 refers to an EDATA address and a value of 7 refers to an FDATA address.
- Generated data for an FDATA segment is placed in Content Records (type 06H) with a **SEG ID** of one. No Segment Definition Record (type 0EH) is generated to define this segment.

14.4 Other Object Formats

In addition to the standard Intel HEX, Motorola SREC or Intel OMF-51 formats, C51ASM supports the following object formats:

14.4.1 Binary

C51ASM can output a pure binary image file. The binary file starts at address zero and continues to the last generated code location. Uninitialized bytes in between the starting and ending addresses are filled with FFH. If CODE and FDATA are unified into a single object, the FDATA data are placed at the next address above the full capacity of the device, unless another location is specified with --mapfseg or $MAPFSEG. Any gap between the CODE and FDATA segments is filled with FFH.

14.4.2 Generic Hexadecimal

The generic hexadecimal format is an ASCII file that specifies addresses and data. Each line starts with a two-byte or three-byte value whose ASCII characters represent the starting hexadecimal address for the subsequent data. Following the address is a list of hexadecimal data bytes, each represented by two ASCII characters. Data values are separated from each other and the address by a space.

Example:

```
0100 21 46 01 36 01 21 47 01 36 00 7E FE 09 D2 19 01
0110 21 46 01 7E B7 C2 00 01 FF 5F 16 00 21 48 01 19
0120 19 4E 79 23 46 23 96 57 78 23 9E DA 3F 01 B2 CA
0130 3F 01 56 70 2B 5E 71 2B 72 2B 73 21 46 01 34 21
```

Section 15

Appendix F - The ASCII Character Set

Table 15-1. ASCII Character Set

HEX	00	10	20	30	40	50	60	70
0	NUL	DLE		0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	NUL	DLE		0	@	P	`	p
C	SOH	DC1	!	1	A	Q	a	q
D	STX	DC2	"	2	B	R	b	r
E	ETX	DC3	#	3	C	S	c	s
F	EOT	DC4	$	4	D	T	d	t

Section 16

Revision History

Revision No.	History
Revision A – October 2010	• Initial Release

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: (+1) (408) 441-0311
Fax: (+1) (408) 487-2600
www.atmel.com
8051@atmel.com

Atmel Asia Limited
Unit 01-5 & 16, 19F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
Tel: (+852) 2245-6100
Fax: (+852) 2722-1369

Atmel Munich GmbH
Business Campus
Packring 4
D-85748 Garching b. munich
GERMANY
Tel: (+49) 89-31970-0
Fax: (+49) 89-3194621

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
JAPAN
Tel: (+81) (3) 3523-3551
Fax: (+81) (3) 3523-7581