STMicrolectronics

MSCI Assembler

UM0145
User manual

Rev 1

June 2005

BLANK

MSCI Assembler

Introduction

The **MSCI Assembler** is an assembly tool based on the ST7 Assembler-Linker, which has been designed to generate the binary code required for implementation of the Mass Storage Controller Interface (MSCI) for your ***ST7267***. The MSCI Assembler does not generate an application executable, but instead outputs the MSCI binary as data that is included in your application for the ST7267 microcontroller.

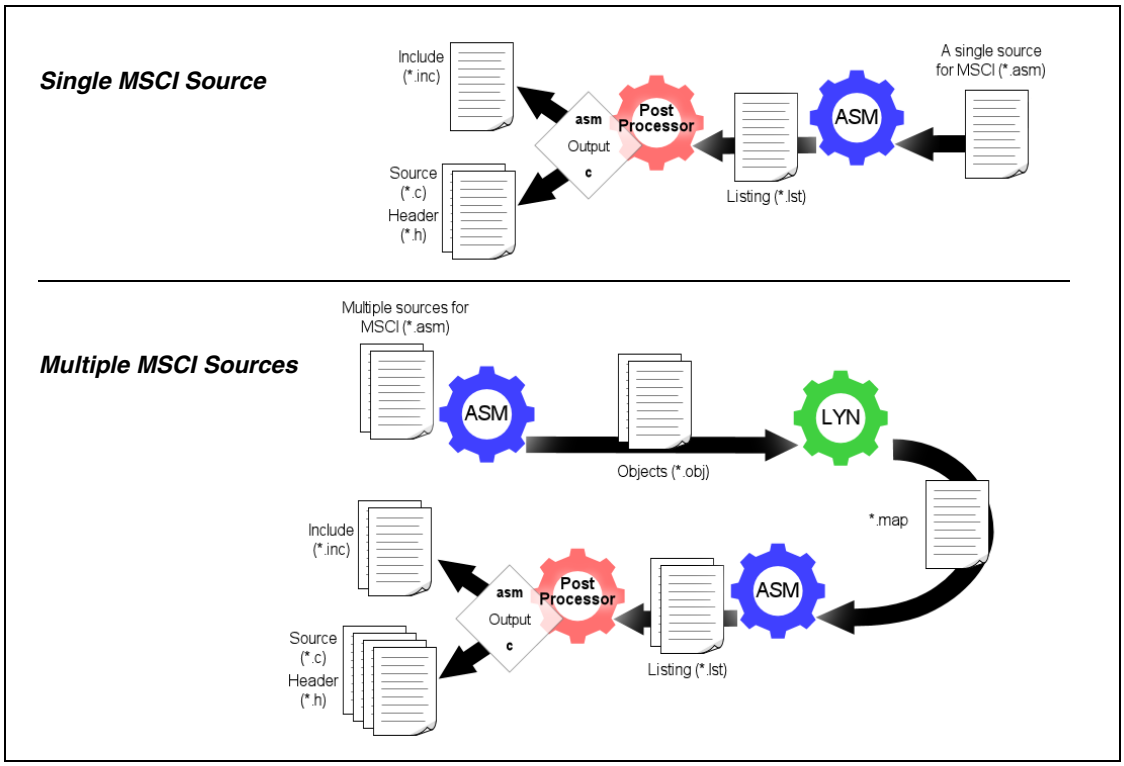
MSCI routines are coded in MSCI assembly language. These routines are assembled and the resulting files are incorporated in the source code for your final ST7 application. Your *ST7267 Datasheet* contains a complete description of the MSCI peripheral with code examples.

The **Assembler (ASM)** produces objects from your MSCI source code (**`*.asm`**) in a first pass. If you only have one MSCI source file, the listing (**`*.lst`**) generated at this stage is used by the Post Processor to produce the final output. Objects are not used.

If you have more than one MSCI source file, the **Linker (LYN)** resolves any external references and then the Assembler produces the final listing (**`*.lst`**) in a second pass.

The **Post Processor** is invoked to convert the listing to the final output (a C source file (**`*.c`**) and a header (**`*.h`**) for C applications, a include (**`*.inc`**) for Assembler).

Figure 1. Schematic overview of the MSCI Assembler toolset



The MSCI Assembler relies on components of the ST7 Assembler-Linker (*ST7 Assembler version 4.48 and Linker version 3.14*) and is delivered and installed as part of the ST7 toolset. In addition to the ST7 Assembler-Linker components, the MSCI Assembler includes the **MSCI.TAB** file – a description file for the MSCI Assembly language, and the **MSCI2ST7.EXE** – the executable that converts the MSCI binary code into ST7 data.

About the user manuals...

This manual provides information about producing the source and/or include files containing the MSCI routines for your application. Here, you will find:

- Instructions for running the MSCI Assembler, Linker and Post Processor
- Descriptions of the MSCI Assembler output

For information on related subjects refer to the following documentation:

ST7267 Datasheet – full description of your ST7's MSCI peripheral and code examples to help you get started.

MSCI Programming Manual – a complete reference to MSCI Assembly language

Host PC system requirements

This tool has been designed to operate on a PC that meets the following:

- One of the following operating systems: Microsoft® Windows® 98, 2000, Millennium, NT® or XP®.
- Intel® Pentium (or compatible) processor with minimum speed of 133 MHz.
- Minimum RAM of 32 MB (64 MB recommended).
- 55 MB of free hard disk space to install all of the ST7 tools.

Getting assistance

For more information, application notes, FAQs and software updates for all the ST microcontroller development tools, check out the CD-ROM or our website:

www.st.com/mcu

For assistance on all ST microcontroller subjects, or for help developing applications that use your microcontroller's MSCI peripheral, refer to the contact list provided in *Product Support* on page 108. We'll be glad to help you.

Contents

1 Getting Started

This section outlines the basic steps to get you started using the MSCI Assembler.

1.1 Understanding the MSCI Assembler

The MSCI Assembler is a specific implementation of a standard assembly toolchain, which allows you to output the files required to implement MSCI routines in an application for your ST7267. This implies some specific constraints that are introduced in this section.

Output format

The binary code resulting from the Assembly of your MSCI sources is output in arrays. A specific use of the segment directive allows you to declare the name of the array containing the binary code (see *Writing the MSCI source code*).

Data format

Data are expected to be 16-bit data. For example:

Correct	Incorrect

`DC.W $1234`
`DC.B $12, $34`
	16-bit data declared on 2 lines can't be used:
	`DC.B $12`
	`DC.B $34`

Instead, the data must be declared as two WORDs:
`DC.W $1234`
`DC.W $5678`
	DC.L (or LONG) can't be used:
	`DC.L $12345678`

This same rule applies to the FCS and STRING directives; only two-byte strings are allowed.

Forbidden directives

The final output is generated from the listing. The following directives that impact the listings are not compatible with generating MSCI output:

- **.nolist** – directive is forbidden otherwise there is no information in the list file.
- **.sall** – directive is forbidden otherwise there is no information for macros in the list file.
- **.tab** – directive is forbidden otherwise Post Builder can't find the different fields of instructions in the list file.

1.2 Installing the MSCI Assembler

The MSCI Assembler is delivered as part of the ST7 toolset. A free installation package is available at **www.st.com/mcu**. To install it:

1. Select ***ST7>ST7 toolchain*** from the main menu of the “Microcontroller Development Tools” CD-ROM, or...

   Run the installation executable that you have downloaded from the internet.

Note: Windows® 2000, NT® and XP®users must have administrator privileges to install certain software components.

After installation, the installation directory should contain the following (*Table 1*):

Table 1. Installed components

MSCI2ST7.exe	MSCI post processor
MSCI.TAB	MSCI description files
ASM.EXE	Assembler
LYN.EXE	Linker
OBSEND.EXE	output file formatter
LIB.EXE	librarian
ST7.TAB	ST7 description files
ASLI.BAT	batch file ASM+LYN+OBSEND
RELEASE_NOTES_AST7-LST7.PDF	release notes

1.3 Writing the MSCI source code

Write your MSCI routines in MSCI assembly language. Your ***ST7267 Datasheet*** provides code examples to help you get started.

To use the MSCI Assembler, your MSCI source files must:

1. Use the file extension **`*.asm`**
2. Specify the directory for the **`msci.tab`** file on the first code line of each file
3. Specify MSCI routines using the *Segment directive*:

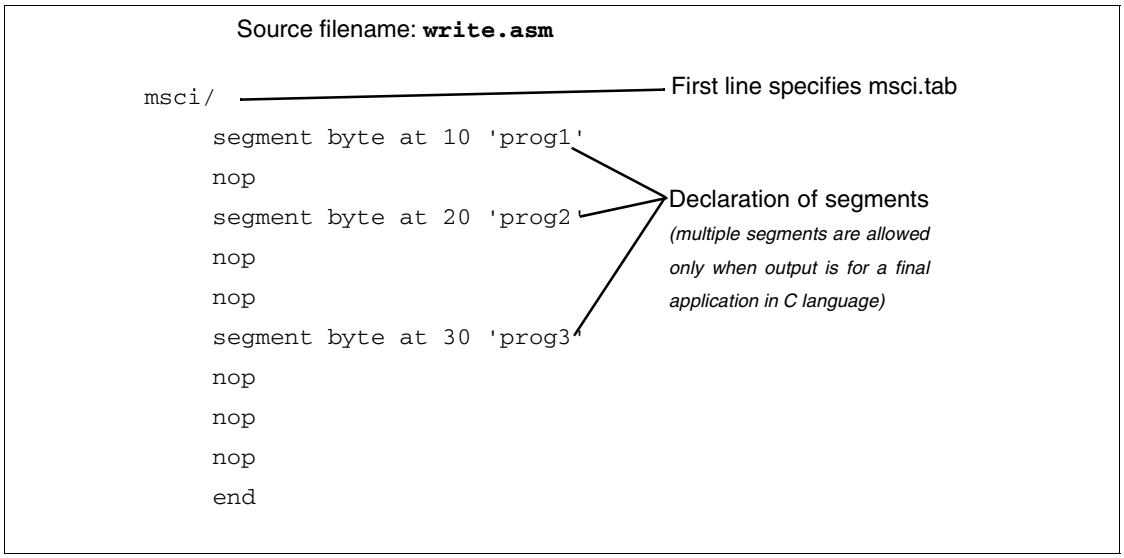
```
[<name>] segment [<align>] [<combine>] '<class>' [cod]
```

When generating output for a C application , the class is used to specify the name of the array containing the binary code of the segment. As a result, a segment can't be used twice with the same class.

Note: Only one segment can be declared for each source if the output is for a final application in Assembly language. An example is shown in Figure 5 on page 36.

Figure 1 provides an example of the format for an MSCI source file that will be used in a final application in C language.

Figure 1. MSCI source for C output



Caution: The following ST7 directives must not be used with the MSCI Assembler: .NOLIST, .SALL, .TAB.

1.4 Running the MSCI Assembler

The MSCI Assembler and Postprocessor can be invoked by entering commands at the command prompt or by using a script such as a batch file (*.bat) that contains the necessary commands.

For a single source file

1. *Invoke the Assembler* to produce a listing (`*.lst`) for the source. Only a single pass and specification of the output of a listing are required.

   Command line format:

   asm <file to assemble> <listing file> <switches> [;]

   Example:

   ```
   asm -li demo
   ```

2. *Invoke the Post Processor* to produce the final C output.

   Command line format:

   msci2st7 <input> <output> [c|asm] [<ST7 offset>]

   Example:

   ```
   msci2st7 demo.lst demo c 0x1100
   ```

For multiple source files

1. *Invoke the Assembler* for a first pass for each source to produce linkable objects.
   Command line format:

   asm <file to assemble> <switches> [;]

   Example:

   ```
   asm demo1
   asm demo2
   ```

2. *Invoke the Linker* for all resulting objects in order to resolve external references and place segments.
   Command line format:

   lyn <.obj>[+<.obj>...], [<.cod file>],[<lib>]

   Example:

   ```
   lyn "demo1+demo2,demo;"
   ```

Note: Libraries are not supported when generating MSCI output.

3. *Invoke the Assembler* for a second pass to produce a listing (*.lst).
   Command line format:

   asm <file to assemble> <listing file> <switches> [;]

   Example:

   ```
   asm demo1 -li -fi=demo
   asm demo2 -li -fi=demo
   ```

4. *Invoke the Post Processor* to produce the final C output.
   Command line format:

   msci2st7 <input> <output>[c|asm] [<ST7 offset>]

   Example:

   ```
   msci2st7 demo1.lst demo1 c 0x1100
   msci2st7 demo2.lst demo2 c 0x1100
   ```

1.5 Incorporating the resulting output in your application

Output for final application in C language consists of a C source file (*.c) and an include file (*.h). The C source file contains arrays corresponding to the MSCI code. External declaration of each array is contained in the header file.

Output for final application in Assembly language consists of an include file (`*.inc`) that contains a word sequence corresponding to the MSCI code.

2 MSCI Addressing Modes

The MSCI instruction set provides a single source-coding model regardless of which components are operands — immediate values, registers such as GP registers (r0, r1, r2, r3) and Data Pointer Registers (dp0, dp1, dp2, dp3), labels or memory referenced by a data pointer. For example, a single instruction, **ld**, initiates register-to-register transfers as well as memory-to-register data movements.

Two-operand instructions are coded with a destination operand appearing in the first position. For example:

```
lab01    ld r0, [dp0]      ; load register with memory indicated by Data Pointer
                             Register
         ld r0,r1          ; load register r0 with value in register r1
```

The MSCI Assembler instruction set incorporates the following different addressing modes:

Table 2. MSCI Addressing Modes

Instruction Set	Addressing Mode	Example
MSCI	Inherent	`nop`
	Immediate	`ld r0,#$55`
	Direct	`ld r0,r1`
	Indirect	`ld r0, [dp0]`
	Absolute	`jrnc loop`
	Bit operation	`bset byte, #5`

All addressing modes are described in full detail, with specific examples, in the *Mass Storage Controller Interface, MSCI 16-Bit Core Programming Manual*. This chapter seeks only to give a brief explanation of the main addressing modes.

Inherent addressing mode

The operand fully specifies all the information required by the CPU to process the operation. No other registers are involved except the status register.

Immediate operands

Immediate operands allow input of a specific value for use with an instruction. They are signaled by the use of a sharp sign (#) before the value.

Examples:

```
lab08    ldl   r0,#1              ; load lower half of r0 with immediate
                                    value 1
lab09    ldh   r0,#1              ; load upper half of r0 with immediate
                                    value 1
```

Immediate load operations can be performed on 8 bits of any register.

Direct and indirect modes

A **direct addressing mode** means that the data byte required to do the operation is found by its memory address, which follows the op-code.

An **indirect addressing mode** means that the data byte required to do the operation is found by its memory address, which is located in the data memory pointer DPx. The pointer address follows the op-code.

The DPx can be automatically incremented or decremented using the AutoINC and AutoDEC instructions.

Absolute mode

This mode can be used with Jump and CALL instructions to modify the PC register value. The new value is in the op-code

Bit operations

This mode is used for bit manipulation. It is a form of direct addressing, but includes an extension indicating the bit in the register that the operation is to be performed on.

3 MSCI Assembler (ASM)

The MSCI Assembler is an assembly tool based on the ST7 Assembler-Linker, which has been designed for assembly of code for MSCI routines.

The MSCI Assembler processes the source code for your MSCI routines and produces a final listing (*.lst). If you have only one MSCI source file, the Assembler outputs directly the final listing required by the *Post Processor* – the linkable objects are not used.

If you have more than one MSCI source file, the Assembler is invoked to generate linkable objects, then the *Linker (LYN)* is invoked. The Assembler is then invoked for each object for a second pass to produce a final listing. The resulting listing is used by the *Post Processor* to generate the files containing the binary code for your MSCI routines that you need to add to your final ST7 application, whether it is in C or Assembly language.

During this process, the Assembler checks for different types of errors. These errors are recorded in an ASCII file called **cbe.err**. (Note that the Linker also writes to this file.) Error messages are explained in *Error Messages*.

This chapter provides information about:

- How to *Invoke the Assembler*
- The structure and contents of *MSCI source files*
- Using *Segment directive* to co-locate sections of code
- Using *Macros* for subroutines in your source code

3.1 Invoke the Assembler

The Assembler command line requires the following arguments:

```
ASM <file to assemble> <listing file> <switches> [;]
```

If any arguments are left out, you'll be prompted for the remaining arguments.

- **`<file to assemble>`** – file name for the source, Assembler assumes that the default file extension **`".ASM"`** is used.
- **`<listing file>`** – the file to which the assembly report is sent. When **`-L`** option the listing is "**`<filename>.LST`**". When **`-FI`** option is specified (when assembling multiple sources), an updated final listing "**`<filename>.LST`**" is produced along with a "**`<filename>.MAP`**" file. (See *LI option* and *FI option*.)
- **`<switches>`** – strings that are specified for replacement when **`-D`** options is specified.

Unless the Assembler is told to create either a listing using the *Options* argument, the listing file will not be created.

3.1.1 Options

Options are always preceded with a minus sign '-'. Upper and lower cases are accepted to define options. Supported options are listed in *Table 3*.

Table 3. Command line options

Option	Function
-SYM	enable symbol table listing
-LI **-LI=<listfile>**	enable pass-2 listing enable listing and specify name of .LST file
-OBJ=<path>	specify .OBJ file
-FI=<mapfile>	specify 'final' listing mode
-D <1> <2>	#define <1> <2>
-I	specify paths for included or loaded files
-M	output make rule
-NP	disable phase errors

SYM option

Description: This option allows the generation of a symbol table.

Format: `ASM <file> -sym`

Example: `ASM prog -sym`
The output is the file `prog.sym`

LI option

Description: Request to generate a complete listing file.
Use the option `-li=<pathname>` to specify the pathname for the generated .LST file.

Format: `ASM <file> -li`
or
`ASM <file> -li=<pathname>`

Example: `ASM prog -li`
The output is the file `prog.lst` in the current directory
`ASM prog -li=obj\prog`
The output is the file `obj\prog.lst`

OBJ option

Description: You can specify the pathname for the generated .OBJ file, using the following option:

Format: `ASM <file> -obj=<pathname>`

Example: `ASM prog -obj=obj\prog`
Forces the AssemblerAssembler to generate the object file `obj\prog.obj`.

FI option

Description: One side effect of using a Linker is that all modules are assembled separately, leaving inter modules' cross-references to be fixed up by the Linker. As a result the Assembler listing file sets all unresolved references to 0, and displays a warning character.

The '`-fi`' option enables you to perform an absolute patch on the desired listing. Therefore, you must have produced a listing file (`.LST`) and linked your application to compute relocations and produce a `.COD` file and a map file.

When you want a full listing to be generated, you must not have made any edits since the last link (otherwise the named map-file would be 'out of date' for the module being assembled). This is not usually a problem since full listings are only needed after all the code has been completed. `-fi` automatically selects a complete listing.

Format: `ASM <file> -fi=<file>.map`
The output `<file>.lst` contains the absolute patches.

*Note: When assembling in '**`-fi`**' mode, the Assembler uses the map file produced by the Linker, and no object files are generated.*

```
Example:        ASM -li ex1                 (produces ex1.lst)
                ASM -li ex2                 (produces ex2.lst)
                LYN ex1+ex2,ex              (produces ex.map, ex.cod)
                                            (see Chapter : Linker on
                                            page 35)
                ASM ex1 -fi=ex.map          (produces new ex1.lst)
                ASM ex2 -fi=ex.map          (produces new ex2.lst)
```

*Note: When using the option **`-fi=<file>.map`**, the Assembler step may fail under certain circumstances:*
***If there are empty segments (Error 73).** To avoid this, comment out any empty segments.*
If you try to assemble a file that has not been used to produce the .map file (Error 73).
***Some EXTERN labels are never used (Warning 80).** To avoid this, comment the unused EXTERN labels out.*

D option

Description: The `-D` option allows you to specify a string that is to be replaced by another during the assembly.

A **blank space** or **=** is required between the string to be replaced and the replacement string. For example `-D <string> 2` is the same as `-D <string>=2`.

It is possible to specify only one argument (`-D <string>`). In which case, `<string>` is replaced with 1.

This is extremely useful for changing the assembly of a module using `#IF` directives, because you can change the value of the `#IF` tests from the Assembler's command line. It means that you can run the Assembler with different `-D` switches on the same source file, to produce different codes.

Format:

```
ASM <file> -D <string> <string>
```

or

```
ASM <file> -D <string>=<string>
```

or

```
ASM <file> -D <string>
```

Example:

```
ASM ex1 -D EPROM 2 -D RAM 3
ASM ex1 -D EPROM=2 -D RAM=3
```

In both cases the string `EPROM` is replaced with `2`. The string `RAM` is replaced with `3`.

```
ASM ex1 -D EPROM
```

In this case `EPROM` is replaced with `1`.

Note: If you specify multiple **-D** *switches, they should always be separated by a space.*

PA option

Description: Request to generate a pass-1 listing. In this listing internal forward references are not yet known. They are marked as undefined with a 'U' in the listing file.

Format:

```
ASM <file> -pa
```

Example:

```
ASM file1 -pa
```

The output file is `file1.lst`

I option

Description: The `-I` option is used to specify the list of search paths for files that are included (with #include) or loaded (with #load).

The different paths can be separated by the ; character and the path list must be enclosed within double quotes. You can also enter multiple include paths by using the`-I` option more than once and separating each with a blank space.

The current working directory is always searched first. After that, the MSCI Assembler searches directories in the same order as they were specified (from left to right) in the command line.

Format:

```
ASM -I="<path1>;<path2>;...;<pathN>" call
```

or

```
ASM -I="<path1>" -I="<path2>"... -I="<pathN>" call
```

Example:

```
ASM -I="include;include2" call
```

or

```
ASM -I="include" -I="include2" call
```

M option

Description: The `-M` option tells the MSCI Assembler to output a rule suitable for make, describing the dependencies to make an object file.

For a given source file, the MSCI Assembler outputs one make rule whose target is the object file name for that source file and whose dependencies are all the included (#include) source files and loaded (#load) binary files it uses. The rule is printed on the standard output.

Format:

```
-M <source file name>
```

Example:

```
ASM -I="include;include2" -M call
```

The output appears on the screen as the rule:

```
call.obj: call.asm include\map.inc include2\map2.inc
include\map3.inc include\code.bin
```

NP option

Description: This option disables the error generation.

Format:

```
ASM <file> -np
```

Example:

```
ASM file1 -np
```

3.2 MSCI source files

Source code for your MSCI routines is written in Assembly language and is saved in ASCII format. A source file has the extension **`*.asm`**. It is made up of lines, each of which is terminated by an end of line character.

To see examples of MSCI routines in Assembly language, refer to your *ST7267 Datasheet*. For a complete reference to the ST7 Assembly language, refer to the *ST7 Programming Manual*.

3.2.1 Assembly source code format

The first line of an assembly source code file is reserved for specifying the *.tab file for the **target processor**, in this case the CPU for the MSCI. You cannot put other instructions or comments in this line.

Use this line to specify the directory location of the msci.tab file. For example, the first line of your source code might look like:

```
c:\st7tools\asm\msci
```

If the file `msci.tab` can't be found in the specified or default directories, then an error is produced and assembly is aborted.

The rest of the source code lines have the following general format:

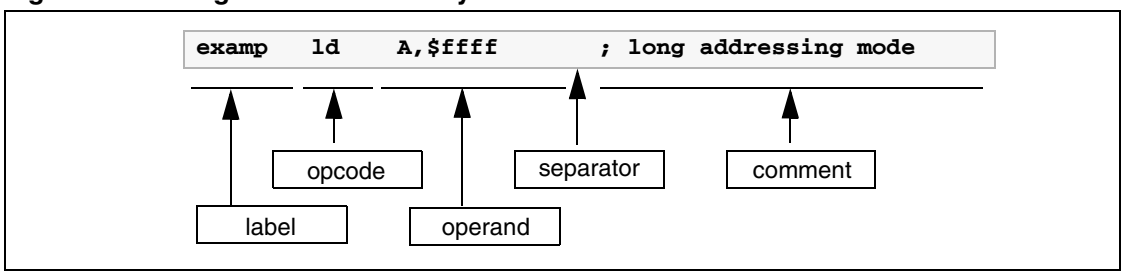
[*Label*[:]]<space>[*Opcode*]<space>[*Operand*]<space>[;*Comment*]

where `<space>` refers to either a SPACE or a TAB character.

All four fields may be left blank, but the **<space>** fields are mandatory unless:

- the whole line is blank, or
- the line begins as a comment, or
- the line ends before the remaining fields.

Figure 2. A single line of Assembly code



3.2.2 Label

A word placed at the beginning of a line with no space preceding it is considered to be a label that can be used by a JUMP instruction. The label can be omitted.

Labels must start in column one. A label may contain up to 30 of any of the following characters:

- Upper Case letters (A-Z)
- Lower case letters (a-z)
- Digits (0-9)
- Underscore (_)

The first letter of a label must be a letter or an underscore. The Assembler is case sensitive.

Upon assembly, any label that exceeds 30 characters is truncated and a warning alerts the user that this has occurred. When truncated, if two of more labels have the same name, a phase inconsistency error is generated.

When labels are defined, the following attributes are defined with the value:

- **Size** (Byte, Word or Long)
- **Relativity** (Linker Relative or Absolute)
- **Scope** (Internally or Externally defined)

Label size

Defining a label's size allows the Assembler to determine what kind of addressing mode to choose even if the value associated with the label is undefined.

The default size of the memory location for a label is word (2 bytes). Whenever the label has no suffix, then the default size is assumed. The directives BYTES， WORDS and LONGS (4 bytes) allow you to change the default.

Regardless of the default size, you can define the size for a specific label by adding a suffix to it: **.b** for byte, **.w** for word and **.l** for long. The suffix is not used when the label is referred to. Using of any suffixes other than .b, .w and .l will result in an error upon assembly.

For example:

```
lab             equ 0             ; word-size label (default)
label1.b        equ 5             ; byte-size label
label2.l        equ 123           ; long label
          segment byte at: 80 'ram'
                bytes             ; force the size of the label to
                                  ; bytes
count           ds.b              ; byte-size label
pointer         ds.w              ; byte-size label with a word-size
                                  ; space reserved at this address
```

Label relativity

There are two sorts of labels: *absolute* labels and *relative* labels.

- *Absolute* labels are usually assigned to constants, such as IO port addresses, or common values used within the program.
- *Relative* labels are defined as (or derived from) an *external* label or a label derived from the position of some program code. They are exclusively used for labels defined within pieces of program or data.

For example:

```
lab             equ 0               ; absolute label 'count'
ioport          equ $8000           ; absolute word label 'ioport'

          segment 'eprom'
start           ld X,#count
                ld A,#'*'
loop            ld ioport,A
```

```
                dec X
                jrne loop
stop            jp stop             ; then loop for ever
```

Only the Linker can sort out the actual address of the code, as the Assembler has no idea how many segments precede this one in the class. At assembly time, labels such as '**start**' or '**loop**' are actually allocated 'blank' values ($0000). These values will be filled later by the Linker. Labels such as '**count**' or '**ioport**', which were defined absolutely will be filled by the Assembler.

Source code lines that have arguments containing relative labels are marked with an 'R' on the listing, showing that they are 'Linker relative'. Segments are discussed in *Section 3.3* on page 24.

Label scope

Often, in multi-module programs, a piece of code will need to refer to a label that is actually defined in another module. To do this, the module that exports the label must declare it PUBLIC, and the module which imports the label must declare it EXTERN. The two directives EXTERN and PUBLIC go together as a pair.

Most labels in a program will be of no interest for other pieces of the program—these are known as 'internal' labels since they are only used in the module where they are defined. Labels are 'internal' by default.

Here are two incomplete example modules that pass labels between them:

```
module 1
        EXTERN _sig1.w          ; import _sig1
        EXTERN _sig2.w          ; import _sig2
        PUBLIC _handlers        ; export _handlers
        segment byte 'P'
_handlers:                      ; define _handlers
        jp _sig1                ; refer to _sig1
        jp _sig2                ; refer to _sig2
        end
module 2
        EXTERN _handlers.w      ; import _handlers (addr. is a word)
        PUBLIC _sig2            ; export _sig2
        segment byte 'P'
_sig2:                          ; define _sig2
        ...
        call _handlers          ; refer to _handlers
        ...
        ret
        end
```

As you can see, module 1 refers to the '**_sig2'** subroutine which is defined in module 2. Note that when module 1 refers to the '**_sig2**' label in an EXTERN directive it specifies a WORD size with the '**.w**' suffix. Because the Assembler cannot look up the definition of '**_sig2**' it has to be told its address size explicitly. It doesn't need to be told relativity: **all external labels are assumed to be relative**.

Absolute labels declared between modules should be defined in an INCLUDE file that is called by all modules in the program; this idea of using INCLUDE files is very important since it can reduce the number of PUBLIC symbols—and therefore the link time—significantly.

Lines in the source code listing which refer to external labels are marked with an X and given 'empty' values for the Linker to fill.

As a short cut, labels may be declared as **PUBLIC** by preceding them with a '**.**' at their definition. If this is done the label name need not be given in a **PUBLIC** directive. For example, the following code fragment declares the label '**lab4**' as PUBLIC automatically:

```
lab3            ld A,#0
                ret
.lab4           nop
                ret
```

3.2.3 Opcode

The second field of characters in a line of code is the Opcode. Opcodes must be preceded by a space or a tab.

The Opcode field may serve three different purposes. It may contain:

- The opcode mnemonic for an assembly instruction,
- The name of a directive,
- The name of a macro to be invoked.

A comprehensive Opcode description can be found in the *ST7 Programming Manual*.

Macros are discussed in *Section 3.4* on page 28.

Directives are discussed in on page 48.

3.2.4 Operand

The third and fourth sets of characters are Operands. The first operand must be proceeded by a space. The second operand must be preceded by a comma (,).

Operands may be any of the following:

- Numbers and addresses,
- String and character constants,
- Program Counter references,
- Expressions.

The following sections explain how to use these types of operands.

Number and address representation

By default, the representation of numbers and addresses follows the MOTOROLA syntax. When you want to use hexadecimal numbers with instructions or labels, they must be preceded by $. When nothing is specified, the default base is decimal.

For example:

```
lab03       equ 10              ; decimal 10
lab04       equ $10             ; hexadecimal 10
            ld A,$ffff          ; long addressing mode
            ld A,#$cb           ; immediate addressing mode
            ld A,#100           ; decimal representation
```

You can change the Motorola format representation by using directives (.INTEL, .TEXAS) to indicate the new setting format. For more information, refer to on page 48.

Caution: Addresses for SEGMENT definition are always given in hexadecimal:
`segment byte at: 100-1FF 'test'`
The segment '`test`' is defined within the 256-511 address range.

Numeric constants and radix

Constants may need special characters to define the radix of the given number.

The Assembler supports the MOTOROLA format by default. INTEL, TEXAS, ZILOG formats are also available if the format is forced by .INTEL .TEXAS or .ZILOG directives. *Table 4* on page 21 shows a summary of these formats.

Note: Decimal constants are always the default, and require no special characters.

Table 4. Numeric constants and radix formats

Format	Hex	Binary	Octal	Current PC
Motorola	$ABCD or &ABCD	%100	~665	* (use `MULT` for MULTIPLY)
Intel	0ABCDh	100b	665o or 665q	$
Texas	>ABCD	?100	~665	$
Zilog	%ABCD	%(2)100	%(8)665	$

String constants

String constants are strings of ASCII characters surrounded by **double quotes**.

ASCII character constants

The Assembler's arithmetic parser also handles ASCII characters in **single quotes**, returning the ASCII of the given character(s).

For example:

```
'A'       $41
'6'       $06
'AB'      $4142
```

Up to 4 characters may be used within a single pair of quotes to give a long constant. The following special sequences are used to denote special characters:

```
'\b'      $7F               backspace
'\f'      $0C               formfeed
'\n'      $0A               linefeed
'\r'      $0D               carriage return
'\t'      $09               tabulation
'\\'      $5C               slash
'\'       $27               single-quote
'\0'      $00               null
'\"'      $22               double-quote
```

Program counter reference

The current value of the program counter (PC) can be specified by an asterisk "*****".

For example:

```
lab05     jra *
```

Expressions and operators

Expressions are numeric values that may be made up of labels, constants, brackets and operators.

Labels and constants have been discussed in previous paragraphs.

Arithmetic brackets are allowed up to 8 nested levels—the curly braces {} are used instead of the common “()” because instructions may use a parenthesis to denote indexed addressing modes.

Operators have 4 levels of precedence. Operators in level #1 (listed in *Table 5*) take precedence over operators in level #2 (listed in *Table 6*), and so on. In each level, operators have same precedence—they are evaluated from left to right.

Table 5. Level 1 operators

Operation	Result, level #1
-a	negated a
a and b	logical AND of A and B
a or b	logical OR of A and B
a xor b	logical XOR of A and B

Table 5. Level 1 operators

Operation	Result, level #1
a shr b	a shifted right b times
a shl b	a shifted left b times
a lt b	1 if a<b, else 0
a gt b	1 if a>b, else 0
a eq b	1 if a=b, else 0
a ge b	1 if a>=b, else 0
a ne b	1 if a unequal b, else 0
high a	a/256, force arg to BYTE type
low a	a MOD 256, force arg to BYTE type
offset a	a MOD 65536, force arg to WORD*16 type
seg a	a/65536, force arg to WORD*16 type
bnot a	invert low 8 bits of a
wnot a	invert low 16 bits of a
lnot a	invert all 32 bits of a
sexbw a	sign extend byte to 16 bits
sexbl a	sign extend byte a to 32 bits
sexwl a	sign extend word to 32 bits

Table 6. Level 2 operators

Operation	Result, level #2
a/b	a divided by b
a div b	a divided by b

Table 7. Level 3 operators

Operation	Result, level #3
a * b	a multiplied by b
a mult b	as above for motorola (character * is reserved)

Table 8. Level 4 operators

Operation	Result, level #4
a-b	a minus b
a+b	a plus b

Operator names longer than one character must be followed by a space character. For example, '`1 AND 2`' is correct, '`1AND2`' is not.

Place the curly braces { } around arithmetic expressions.

Also, always use curly braces at the top-level, when defining a numeric expression. Not doing so may produce unexpected results.

```
Wrong syntax:
            #define SIZE 128
            DS.W SIZE+1           ; Wrong, syntax error
            #IF SIZE eq 1         ; Wrong, same as #IF SIZE
            #ENDIF

Correct syntax:
            #define SIZE 128
            DS.W {SIZE+1}         ; OK
            #IF {SIZE eq 1}       ; OK
            #ENDIF
```

3.2.5 Comment

Comments are preceded by a semicolon (;). Characters following a semicolon are ignored by the Assembler.

3.3 Segment directive

Segmentation is a way of 'naming' areas of your code and making sure that the Linker collates areas of the same name together in the same memory area, whatever the order of the segments in the object files. When outputting for a final application in C language, you can declare segments for each MSCI routine in a source file, up to 128 different segments per source. However, when outputting for a final application in Assembly language, only one segment is allowed in each MSCI source file.

The segment directive itself has four arguments, separated by spaces:

[<*Name*>] SEGMENT [<*Align*>] [<*Combine*>] '<class>' [*Cod*]

- **[<name>]** – (*optional*) name of the segment up to 12 characters.
- **[<align>]** – the threshold at which each segment must start.
- **[<combine>]** – specifies how segments in a class are treated when given assigned to memory.
- **'<class>'** – (*mandatory*) name of the class up to 30 characters.
- **[cod]** – allows specification of '**.COD**' file that segments are sent to. If **[cod]** field is omitted then the segment is sent to the default file `<output>.cod`.

For example:

File 1:

```
st7/
                    BYTES
                    segment byte at: 80-FF 'RAM0'
counter.b           ds.b                        ; loop counter
```

```
address.b           ds.w                        ; address storage
                    ds.b 15                     ; stack allocation
stack               ds.b                        ; stack grows downward


                    segment byte at: E000-FFFF 'eprom'
                    ld A,#stack
                    ld S,A                      ; init stack pointer
                    end

File 2:
st7/
                    segment 'RAM0'
serialtemp          ds.b
serialcou           ds.b

                    WORDS
                    segment 'eprom'
serial_in           ld A,#0
                    end
```

In the preceding example, File 1 and File 2 are two separate modules belonging to the same program. File 1 introduces two classes: **'RAM0'** and **'eprom'**.

Class names may be any names you choose up to 30 characters. The first time a class is used, you have to declare the default alignment, the start and the end addresses of the class, and of course, the name of the class. Users generally specify a new class for each 'area' of their target system.

In the examples above, the user has one class for the 128 bytes of on-chip RAM from **0080** to **00FF** (**'RAM0'**) and another for the **'eprom'**.

The code is stored from **E000** to **FFFF** (**'eprom'**). You have to supply all this information the very first time you use a new class, otherwise only the class-name is necessary, as in File 2.

3.3.1 Name

The **<name>** argument is optional; it can contain a name of up to 12 characters. If it does, then all segments with the same name are grouped together within their class, in the order that new names are defined.

3.3.2 Align

The **<align>** argument defines the threshold on which each segment must start. The default is the alignment specified at the introduction of the class (if none is specified in the class

introduction then `para` alignment is assumed), although the alignment types described in *Table 9* are allowed to be specified overriding the default.

Table 9. Alignment types

Type	Description	Examples
byte	Any address	
word	Next address on boundary	**1001->1002**
para	Next address on 16-byte boundary	**1001->1010**
64	Next address on 64-byte boundary	**1001->1040**
128	Next address on 128-byte boundary	**1001->1080**
page	Next address on 256-byte boundary	**1001->1100**
long	Next address on 4-byte boundary	**1001->1004**
1k	Next address on 1k-byte boundary	**1001->1400**
4k	Next address on 4K-byte boundary	**1001->2000**

Looking back to our example on *page 24*, you should now be able to see that the **'RAM0'** class will allocate **80** to **counter**, **81** to **address**, **92** to **stack** in FILE1, and when the Linker meets the segment in FILE2 of the same class, **serialtemp** will be allocated **93**, and **serialcou 94**. The same processing happens to the two **'eprom'** class segments—the second, in FILE2, will be tacked on to the end of the first in FILE1. If the FILE2 **'eprom'** class segment had specified, say, the **long** align type instead of the default **byte**, then that segment would have been put on the next long-word boundary after the end of the FILE1 **'eprom'** class segment.

3.3.3 Combine

The **<combine>** argument tells the Assembler and Linker how to treat the segment. There are three types to handle it:

Table 10. Combine types

Type	Description
at:X[-Y]	Starts a new class from address X [to address Y]
common	All common segments that have the same class name will start at the same address. This address is determined by the Linker.
<none>	Follows on from end of last segment of this class.

Note: The `at`-type `<combine>` must be used at the introduction of a class, only once.

The **at**-type **<combine>** must have one argument: the start address of the class, and may optionally be given the end address (or limit) of the class too. If given, the Linker checks that no items in the class have gone over the limit address; if this does occur, a warning is issued at link time. The hexadecimal numbers X and Y should not have radix specifiers.

All **common**-type **<combine>** segments that have the same class name will start at the same address. The Linker keeps track of the longest segment. **common** segments can be used for sharing data across different applications.

For example:

```
st7/
dat1        segment byte at: 10 'DATA'
            ds.w
com1        segment common 'DATA'
.lab1       ds.w 4
com1        segment common 'DATA'
.lab2       ds.w 2
com2        segment common 'DATA'
.lab3       ds.w
com2        segment common 'DATA'
.lab4       ds.w 2
dat2        segment 'DATA'
.lab5       ds.w 2
            end
```

The values for labels **lab1**, **lab2**, **lab3**, **lab4,** and **lab5** are **12**, **12**, **1A**, **1A** and **1E,** respectively.

Note: Since you can't specify both ***at*** *and* ***common*** *combines simultaneously, the only way to specify the exact location of commons is to insert an empty* ***at*** *combine segment before the first* ***common*** *declaration.*

For example:

```
com1        segment byte at: 10 'DATA'
com1        segment common 'DATA'
            ...
com1        segment common 'DATA'
            ...
```

3.3.4 Cod

The last field of a **SEGMENT** directive controls where the Linker places the code for a given class. When introducing a class, if this field is not specified, the code for this class will be sent to the normal, default **.COD** file by the Linker. If the **[cod]** file is given a number between **0** and **9** then all code generated under the class being introduced will be sent to a different '**.COD**' file by the Linker.

If the Linker produces a file called '**prog.cod**', for example, then all code produced under classes with no **[cod]** field will go into that file, as normal.

If one class is introduced with a **[cod]** field of **1**, though, then all code produced under that class is sent instead to a file **prog_1.cod**. The code produced under the other classes is sent on as usual to **prog.cod**.

Using this scheme, you can do bank switching schemes quickly and directly, even when multiple EPROMs share the same addressing space. Simply allocate each EPROM class of its own, and introduce each class with a different **[cod]** field. This will result in the Linker collating EPROM's contents into a different .COD file for you to OBSEND independently.

For example:

```
segment byte at:8000-BFFF 'eprom1' 1
segment byte at:8000-BFFF 'eprom2' 2
```

3.4 Macros

Macros are **assembly-time subroutines**. When you call an execution-time subroutine you have to go through several time-consuming steps: loading registers with the arguments for the subroutine, having saved and emptied out the old contents of the registers if necessary, pushing registers used by the subroutine (with its attendant stack activity) and returning from the subroutine (more stack activity) then popping off preserved registers and continuing.

Although macros don't get rid of all these problems, they can go a long way toward making your program execute faster than using subroutines—at a cost. The cost is program size.

Each time you invoke a macro to do a particular job, the whole macro assembly code is inserted into your source code.

This means there is no stacking for return addresses—your program just runs straight into the code; but it's obviously not feasible to do this for subroutines above certain size.

The true use of macros is in small snippets of code that you use repeatedly— perhaps with different arguments—which can be formalized into a 'template' for the macros' definition.

3.4.1 Defining macros

Macros are defined using three directives: **MACRO, MEND** and **LOCAL**.

The format is:

```
<macro-name>MACRO  [parameter-1][, parameter-2 ...]
   [LOCAL] <label-name>[, label-name ...]]
   <body-of-macro>
   MEND
```

For example:

```
add16                    MACRO first,second,result
                         ld A,first
                         adc A,second
                         ld result,A
                         MEND
```

The piece of code in the example might be called by:

```
   add16 index,offset,index
```

which would add the following statements to the source code at that point:

```
ld A,index
adc A,offset
ld index.X,A
```

The formal parameters given in the definition have been replaced by the actual parameters given on the calling line. These new parameters may be expressions or strings as well as label names or constants. Because they may be complex expressions, they are bracketed when there is any extra numeric activity; this is to make sure they come out with the precedence correctly parsed.

Macros do not need to have any parameters. You may leave the MACRO argument field blank (and, in this case, give no parameters on the calling line).

There is one further problem; because a macro may be called several times in the same module, any labels defined in the macro will be duplicated. The **LOCAL** directive gets around this problem:

For example:

```
getio                   MACRO
                        LOCAL loop
loop                    ld A,$C000
                        jra loop
                        MEND
```

This macro creates the code for a loop to await IO port at **$C000** to go low. Without the **LOCAL** directive, the label '**loop**' would be defined as many times as the macro is called, producing syntax errors at assembly time.

Because it's been declared **LOCAL** at the start of the **MACRO** definition, the Assembler takes care of it. Wherever it sees the label '**loop**' inside the macro, it changes the name '**loop**' to '**LOCXXXX**' where **XXXX** is a hex number from **0000** to **FFFF**.

Each time a local label is used, **XXXX** is incremented. So, the first time the **getio** macro is called, '**loop**' is actually defined as '**LOC0**', the second time as '**LOC1**' and so on, each of these being a unique reference name. The reference to '**loop**' in the 'if' statement is also detected and changed to the appropriate new local variable.

The directives in *Table 11* are very useful, in conjunction with macros:

Table 11. Some useful directives

Directive	Usage
#IFB	To implement macro optional parameters.
#IFDEF	To test if a parameter is defined.
#IFLAB	To test if a parameter is a label.
#IFIDN	To compare a parameter to a given string.

Caution: The directive .sall must not be used with the MSCI Assembler. If used, no information for macros will be written to the final listing that is required by the Post Processor.

3.4.2 Parameter substitution

The Assembler looks for macro parameters after every space character. If you want to embed a parameter, for example, in the middle of a label, you must precede the parameter name with an ampersand '**&**' character, to make the parameter visible to the preprocessor. For example, if we have a parameter called '**param**'.,

```
dc.w param
```

it works as expected, but the ampersand is necessary on:

```
label&param:nop
```

```
label&param&_&param:nop
```

Otherwise '**labelparam**' would be left as a valid label name; If the macro parameter '**param**' had the value '**5**', then '**label5**' and '**label5_5**' would be created.

4 Linker (LYN)

If you have more than one MSCI source file, the *MSCI Assembler (ASM)* is invoked to generate linkable objects, then the Linker is invoked. The linking process resolves all external references. If a referenced label is not defined as **PUBLIC**, an error is detected. The Linker also checks the type of relocation to do, places the segments according to your mapping and checks if any of them is overrun.

After linking, the Assembler is invoked for a second pass for each object to produce a final listing. The resulting listing is used by the *Post Processor* to generate the files containing the binary code for your MSCI routines that you need to add to your final application whether it is in C or Assembly language.

This chapter provides information about:

- How to *Invoke the Linker*
- *Input and output files* , including OBJ, COD and MAP files

4.1 Invoke the Linker

The Linker needs the following arguments:

```
LYN <.OBJ file>[+<.OBJ file>...],[<.COD file>],[<lib>]
```

If all or any arguments are left out of the command line, you'll be prompted.

- **<.OBJ file>** – a list of all the object files that form your program. The **.OBJ** suffix may be left out, and if more than one is specified they should be separated by '**+**' characters, for example **game+scores+keys** would tell the Linker to link '**game.obj**', '**scores.obj**' and '**key.obj**'. Object file path names should not include '**-**' or '**;**' characters. Character '**.**' should be avoided, except for suffixes.
- **[<.COD file>]** – *(optional)* specify the name of the **.COD** file if desired.
- **[<lib>]** – it is not possible to use libraries when generating MSCI output as the sources must be assembled each time the output is produced.

4.2 Input and output files

At the end of a successful link, one or more **.OBJ** files will have been combined into a single **.COD** file. A **.MAP** file will have been produced, containing textual information about the segments, classes and external labels used by the **.OBJ** module(s). Finally a compact **.SYM** file is generated, containing all **PUBLIC** symbols found in the link with their final values.

4.2.1 The composition of the .OBJ files

The .OBJ files produced by the Assembler contain an enormous amount of overhead, mostly as coded expressions describing exactly what needs to go into the 'blank spaces'. The Linker contains a full arithmetic parser for working out complex expressions that include external labels: This means (unlike most other Assemblers) there are few restrictions on where external labels may appear.

The Assembler also includes line-number information with the **.OBJ** file, connecting each piece of generated object code with a line number from a given source file.

OBJ files also contain 'special' markers for handling **SKIP** and **DATE** type directive.

4.2.2 The composition of the .COD files

.COD files, on the other hand, contain very little overhead; there are six bytes per segment that describe the start address and length of that segment. Besides that, the rest of the code is in its final form. A segment of zero length marks the end of the file.

4.2.3 Reading a mapfile listing

The Linker also generates files with the suffix **.SYM** and **.MAP** in addition to the **.COD** file we have already discussed. The **.SYM** file contains a compact symbol table list suitable with the debuggers and simulators.

The **.MAP** file listing shows three important things: a table of segments with their absolute address, a table of all classes in the program, and a list of all external labels with their true values, modules they were defined in and size.

Here is an example **MAPFILE**, where one of the class, **ROM**, has gone past its limit, overwriting (or more correctly, having part of itself overwritten by) **VECTORS**.

The **[void]** on some segments in the segment list says that these segments were not used to create object code, but were used for non-coding-creating tasks such as allocating label values with **ds.b** etc. The number in straight brackets on the segment as true address list shows how many segments 'into' the module this segment is, i.e., the 1St, 2nd etc. of the given module. The first x-y shows the range of addresses. The **def (line)** field on the external labels list shows the source code file and line number that his label was defined in. The number at the start of each class list line is the cod-file that the class contents were sent to (default is **0**).

Segment Address List:

```
prog [1]          10-       86        0-        6         'RAM0' [void]
prog [2]          88-       278       100-      138       'RAM1' [void]
main [1]          8-        563       8000-     875B      'eprom'
prog [4]          282-      889       C000-     C508      'rom'
main [2]          568-      1456      C509-     F578      'rom'
monitor [1]       8-        446       F579-     FFF9      'rom'
monitor [2]       448-      467       FFEE-     FFFF      'vectors'
```

Class List:

```
0      'RAM0'       byte from 0 to 78 (lim   FF) 45% D
0      'RAM1'       byte from 100 to 138 (lim 27F) 50% D
0      'eprom'      byte from 8000 to 875B (lim BFFF) 21% C
0      'rom'        byte from C000 to FFF9 (lim FFDF) C*Overrun*
0      'vectors'    byte from FFEE to FFFF (lim FFFF) 100% D
```

External Label List:

```
Symbol Name             Value           Size        Def(line)

char                    64              BYTE        game.obj(10)
char1                   66              BYTE        game.obj(11)
label                   ABCD            WORD        game.obj(25)

3 labels
```

The **external label list** only includes labels that were declared **`PUBLIC`**: labels used internally to the module are not included. This table is most useful for debugging purposes, since the values of labels are likely to be relocated between assemblies. The labels are given in first-character-alphabetic order.

5 Post Processor

The final listing (*.lst) produced by the *MSCI Assembler (ASM)*, is used by the *Post Processor* to generate the files containing the binary code for your MSCI routines that you need to add to your final application whether it is in C or Assembly language.

This chapter provides information about:

- How to *Invoke the Post Processor*
- *Post Processor output* for applications in C or Assembly language

5.1 Invoke the Post Processor

The Post Processor command line requires the following arguments:

```
msci2st7 <input file> <output file> [c|asm] [<ST7 offset>]
```

If any or all the arguments are left out of the command line, you'll be prompted for the remaining arguments.

- **<input file>** – file name of the final listing generated by the Assembler, the Post Processor requires that the suffix ***.lst** be used.
- **<output file>** – the output file name. No suffix is specified as this will be determined by the type of output selected (**c** or **asm**). The Post Processor produces ***.c** and ***.h** files for applications in C language and a ***.inc** file for applications in Assembly language.
- **[c|asm]** – the type of output to produce: **c** for applications in C language, **asm** for applications in Assembly language.
- **[<ST7 offset>]** – default is 0.

5.2 Post Processor output

The Post Processor uses the listing file to output the MSCI binary as data which is included in your ST7 application. It generates different output files depending on whether they are to be included in an ST7 application in C, or Assembly language. The output for applications in C and Assembly is described in the following sections.

For both C and Assembly output, the Post Processor outputs the MSCI binary in tables. After each instruction code, a comment is added followed by the hexadecimal ST7 address, the hexadecimal MSCI address in parentheses and the MSCI instruction from the list file. Because of the possible use of macros, the list file may not look exactly like the source file.

5.2.1 Output for final application in C language

When **c** is specified in the Post Processor command line, two files are generated – `<filename>.c` and `<filename>.h`.

For each segment declared in the MSCI source file, a constant array of elements of type unsigned char is declared and initialized in the C source file. Arrays are initialized with the byte sequence corresponding to the MSCI code. The name of the array is the class of the segment with the keyword “`array`”.

For each array, the header file will contain its external declaration and the definitions of its length, its start address taking into account the ST7 offset and its execution address (its MSCI address). The ST7 offset will be the same for each array.

Figure 3 and *Figure 4* show the C output for the example MSCI source (`write.asm`) shown in *Figure 1* of the *Getting Started*. The Post Processor command line in this case is: `msci2st7 write.lst write c`

Figure 3. Output C source file for `write.asm`

Source filename: **`write.asm`** Output filename: **`write.c`**

```
const unsigned char prog1Array[] = {
         0x00, 0x00 // 0020 (0010)           nop
};

const unsigned char prog2Array[] = {
         0x00, 0x00 // 0040 (0020)           nop
        ,0x00, 0x00 // 0042 (0021)           nop
};

const unsigned char prog3Array[] = {
         0x00, 0x00 // 0060 (0030)           nop
        ,0x00, 0x00 // 0062 (0031)           nop
        ,0x00, 0x00 // 0064 (0032)           nop
};
```

Figure 4. Output header file for `write.asm`

Source filename: **`write.asm`** Output filename: **`write.h`**

```
extern const unsigned char prog1Array[];
#define PROG1ARRAYSTART 0x0020
#define PROG1ARRAYEXEC 0x0010
#define PROG1ARRAYLENGTH0x0002// 2

extern const unsigned char prog2Array[];
#define PROG2ARRAYSTART 0x0040
#define PROG2ARRAYEXEC 0x0020
#define PROG2ARRAYLENGTH0x0004// 4

extern const unsigned char prog3Array[];
#define PROG3ARRAYSTART 0x0060
#define PROG3ARRAYEXEC 0x0030
#define PROG3ARRAYLENGTH0x0006// 6
```

5.2.2 Output for final application in Assembly language

When **asm** is specified in the Post Processor command line, one file is generated for each MSCI source file–`<filename>.inc` . This include file contains the word sequence corresponding to the MSCI code.

Figure 5 shows an example of an MSCI source file (**call_ret.asm**) for use in a final project in Assembly language (*Note the declaration of only one segment for this source. If more than one segment is declared in a source, the Assembler will return and error.*).

Figure 5. MSCI source for ASM output

Source filename: **call_ret.asm**

```
msci/
        segment byte at 10 'test'
        jp begin_it
to_one  ldhr3,#$00
        ldlr3,#$01
        ret
begin_itldhr3,#$ff
        ldlr3,#$ff
        callto_one
        nop
        nop
        end
```

Figure 6 shows the ASM output for this source file when the Post Processor command line is:

```
msci2st7 call_ret.lst call_ret asm 0x100
```

Figure 6. Output include file for call_ret.asm

Source filename: **call_ret.asm** Output filename: **call_ret.inc**

```
call_ret.inc
        DC.W $4014 ; 0120 (0010)           jp      begin_it
        DC.W $e300 ; 0122 (0011) to_one    ldh     r3,#$00
        DC.W $c301 ; 0124 (0012)           ldl     r3,#$01
        DC.W $0001 ; 0126 (0013)           ret
        DC.W $e3ff ; 0128 (0014) begin_it  ldh     r3,#$ff
        DC.W $c3ff ; 012a (0015)           ldl     r3,#$ff
        DC.W $4811 ; 012c (0016)           call    to_one
        DC.W $0000 ; 012e (0017)           nop
        DC.W $0000 ; 0130 (0018)           nop
```

Appendix A Assembler Directives

A.1 Introduction

Each directive has a section to itself, and an entry in the index. The name of the directive that is used in the second field, is given in the heading at the top of the section. The next line shows arguments allowed (if any) for this directive. This is followed by a description of the directive and the format and nature of the argument specified in the previous section. The last section gives one or more examples of the directive in use.

All the directives must be placed in the second, OPCODE, field, with any arguments one tab away in the argument field.

The following directives are specific to the MSCI Assembler:

- *.OFFSETST7*
- *.ORG*

Caution: The following ST7 directives **must not** be used with the MSCI Assembler: *.NOLIST, .SALL, .TAB.*

A.2 .BELL

Purpose:	Ring bell on console.
Format:	`.BELL`
Description:	This directive simply rings the bell at the console; it can be used to signal the end of pass-1 or pass-2 with #IF1 or #IF2. This directive does not generate assembly code or data.
Example:	`.BELL`
See Also:	

A.3 BYTE

Purpose:	Define byte in object code.
Format:	`BYTE <exp or "string">,[,<exp or "string">...]`
Description:	This directive forces the byte(s) in its argument list into the object code at the current address. The argument can have complex expressions, which may include external labels. If the argument is an expression with a value greater than 255, then the lower 8 bits of the expression are used and no errors are generated. String arguments (generally used for defining data tables) are surrounded by double quotes: theses are translated into ASCII and processed byte by byte. Synonymous with STRING and DC.B.

Example:	`BYTE 1,2,3 ; generates 01,02,03`
	`BYTE "HELLO" ; generates 48,45,4C,4C,4F`
	`BYTE "HELLO",0 ; generates 48,45,4C,4C,4F,00`

See Also:	DC.B, STRING, WORD, LONG, DC.W, DC.L

A.4 BYTES

Purpose:	Label type definition where type = byte.
Format:	`BYTES`
Description:	When a label is defined, 4 separate attributes are defined with it: scope (internally or externally defined), value (actual numerical value), relativity (absolute or relative), and length (BYTE, WORD and LONG). All attributes, except length, are defined explicitly prior to or at the definition. The label can be forced to a certain length by giving a dot suffix—eg. 'label.b' forces it to be byte length. Label length can also have a default state: eg. The default can be set to WORD at the start of the assembly, but may be changed to the appropriate length using BYTES, WORDS or LONGS.
Example:	`BYTES` `lab1 EQU 5 ; byte length for lab1`
See Also:	LONGS, WORDS

A.5 CEQU

Purpose:	Equate pre-existing label to expression.
Format:	`label  CEQU <exp>`
Description:	This directive is similar to EQU, but allows to change the label's value. Used in macros and as counter for REPEAT / UNTIL.
Example:	`lab1    CEQU {lab1+1} ; inc lab1`
See Also:	EQU, REPEAT, UNTIL

A.6 .CTRL

Purpose:	Send control codes to the printer.
Format:	`.CTRL <ctrl>[,<ctrl>]...`
Description:	This directive is used to send printing and non printing control codes to the selected listing device. It's intended for sending control codes to embolden or underline, etc. areas of listing on a printer. The arguments are sent to the listing device if the listing is currently selected. This directive does not generate assembly code or data.
Example:	`.CTRL 27,18`
See Also:	.LIST, .NOLIST, .BELL

A.7 DATE

Purpose:	Define 12-byte ASCII date into object code.
Format:	`DATE`
Description:	This directive leaves a message for the Linker to place the date of the link in a 12-byte block left available by the Assembler at the position of the DATE directive. Every link will leave its date in the object code, allowing automatic version control. The date takes the form (in ASCII) DD_MMM_YYYY; for example 20 JUL. 2005. The date is left for the Linker to fill instead of the Assembler since the source code module containing the DATE directive may not be reassembled after every editing session and it would be possible to lose track.
Example:	`DATE`
See Also:	

A.8 DC.B

Purpose:	Define byte(s) in object code.
Format:	`DC.B <exp or "string">,[,<exp or "string">]`
Description:	This directive forces the byte(s) in its argument list into the object code at the current address. The argument may be composed of complex expressions, which may even include external labels. If the argument was an expression and had a value greater than 255 then the lower 8 bits of the expression are used and no errors are generated. String argument(s) must be surrounded by double-quotes: these are translated into ASCII and processed byte by byte. It's generally used for defining data tables. Synonymous with **BYTE** and **STRING**.

Example:	`DC.B 1,2,3 ; generates 01,02,03`
	`DC.B "HELLO" ; generates 48,45,4C,4C,4F`
	`DC.B "HELLO",0 ; generates 48,45,4C,4C,4F,00`

See Also:	

A.9 DC.W

Purpose:	Define word(s) in object code.
Format:	`DC.W<exp>[, <exp>...]`
Description:	This directive forces the word(s) in its argument list into the object code at the current address. The arguments may be composed of complex expressions, which may even include external labels. If the argument was an expression and had a value greater than FFFF then the lower 16 bits of the expression are used and no errors are generated. DC.W sends the words with the most significant byte first. It's generally used for defining data tables. Synonymous with **WORD**, except that DC.W places the words in High / Low order.
Example:	`DC.W 1,2,3,4,$1234 ;0001,0002,0003,0004,1234`
See Also:	DC.B, BYTE, STRING, WORD, LONG, DC.L

A.10 DC.L

Purpose:	Define long word(s) in object code.
Format:	`DC.L <exp>[,<exp>...]`
Description:	This directive forces the long word(s) argument list into the object code at the current address. The arguments may be composed of complex expressions, which may even include external labels. If the argument was an expression and had a value greater than FFFFFFFF then the 32 bits of the expression are used and no errors are generated. **DC.L** sends the words with the most significant byte first. It's generally used for defining data tables. Synonymous with **LONG**, except that DC.L stores the long-words in High / Low order.
Example:	`DC.L 1,$12345678 ; 0000,0001,1234,5678` `LONG 1,$12345678 ; 0100,0000,7856,3421`
See Also:	DC.B, DC.W, BYTE, STRING, WORD, LONG

A.11 #DEFINE

Purpose:	Define manifest constant.
Format:	`#DEFINE <CONSTANT ID> <real characters>`
Description:	The benefits of using labels in Assembler level programming are obvious and well known. Sometimes, though, values other than the straight numerics allowed in labels are used repeatedly in programs and are ideal candidates for special labelling. The #DEFINE directive allows you to define special labels called 'manifest constants'. These are basically labels that contain strings instead of numeric constants. During the assembly, wherever a manifest ID is found in the source code, it is replaced by its real argument before the assembly proceeds. The #DEFINE is not the definition of a label, so a space must precede the declaration. The number of defines that the Assembler can manage is limited to 4096. However, this depends on the number of characters in the statements. Depending on their length, you may reach this limit sooner.
Example:	`#define value 5` `ld a,#value ; ld a,#5`
See Also:	

A.12 DS.B

Purpose:	Define byte space in object code.
Format:	`DS.B [optional number of bytes]`
Description:	This directive is used to 'space out' label definitions. For example let's say we need a set of byte-sized temporary storage locations to be defined in RAM, starting at address $4000. We could write: segment byte at 4000 'RAM' temp1 equ $4000 temp2 equ $4001 which would work fine, however, we recommend you to write: segment byte at 4000 'RAM' temp1 DS.B temp2 DS.B which does the same job. The advantage is that the **PC** increments automatically. There are two other types of DS instructions available for doing WORD and LONG length storage areas: DS.W and DS.L. Note that the areas in question are not initialized to any value; it's merely a way of allocating values to labels. The optional argument specifies how many bytes to allocate; default is 1. Because no code is generated to fill the space, you are not allowed to use DS.B in segments containing code, only for segments with data definitions.
Example:	`labl DS.B`
See Also:	DS.W, DS.L

A.13 DS.W

Purpose:	Define word space in object code.
Format:	`DS.W [optional number of words]`
Description:	This directive is used to 'space out' label definitions. For example let's say we need a set of word-sized temporary storage locations to be defined in RAM, starting at address $4000. We could write: segment byte at 4000 'RAM' temp1 equ $4000 temp2 equ $4002 which would work fine, however, we recommend you to write: segment byte at 4000 'RAM' temp1 DS.W temp2 DS.W which does the same job. The advantage is that the **PC** increments automatically. There are two other types of DS instructions available for doing BYTE and LONG length storage areas: DS.B and DS.L. Note that the areas in question are not initialized to any value; it's merely a way of allocating values to labels. The optional argument specifies how many bytes to allocate; default is 1. Because no code is generated to fill the space, you are not allowed to use DS.W in segments containing code, only for segments with data definitions.
Example:	`labl DS.W`
See Also:	DS.B, DS.L

A.14 DS.L

Purpose:	Define long space in object code.
Format:	`DS.L [optional number of long words]`
Description:	This directive is used to 'space out' label definitions. For example let's say we need a set of long-word-sized temporary storage locations to be defined in RAM, starting at address $4000. We could write: segment byte at 4000 'RAM' temp1 equ $4000 temp2 equ $4004 which would work fine, however, we recommend you to write: segment byte at 4000 'RAM' temp1 DS.L temp2 DS.L which does the same job. The advantage is that the **PC** increments automatically. There are two other types of DS instructions available for doing BYTE and WORD length storage areas: DS.B and DS.W. Note that the areas in question are not initialized to any value; it's merely a way of allocating values to labels. The optional argument specifies how many bytes to allocate; the default is 1. Because no code is generated to fill the space, you are not allowed to use DS.L in segments containing code, only for segments with data definitions.
Example:	`labl DS.L`
See Also:	DS.B, DS.W

A.15 END

Purpose:	End of source code.
Format:	`END`
Description:	This directive marks the end of the assembly on the main source code file. If no **END** directive is supplied in a source-code file then an illegal EOF error will be generated by the Assembler. Include files do not require an END directive.
Example:	`END`
See Also:	

A.16 EQU

Purpose:	Equate the label to expression.
Format:	`label EQU <EXPRESSIONS>`
Description:	Most labels created in a program are attached to a source code line that generates object code, and are used as a target for jumps or memory references. The rest are labels used as 'constants', used for example, to hold the IO port number for the system keyboard: a number that will remain constant throughout the program. The **EQU** directive allocates the value, segment type and length to the label field. The value is derived from the result of the expression, the relativity (absolute or segment-relative derived from the most recent segment), the length is BYTE, WORD or LONG, derived from the size default (starts off as WORD and may be changed by directives **BYTES**, **WORDS** or **LONGS**).
Example:	`labl  END 5`
See Also:	

A.17 EXTERN

Purpose:	Declare external labels.
Format:	`EXTERN`
Description:	When your program consists of several modules, some modules need to refer to labels that are defined in other modules. Since the modules are assembled separately, it is not until the link stage that all the necessary label values are going to be known. Whenever a label appears in an **EXTERN** directive, a note is made for the Linker to resolve the reference. Declaring a label external is just a way of telling the Assembler not to expect that label to be defined in this module, although it will be used. Obviously, external labels must be defined in other modules at link stage, so that all the gaps left by the Assembler can be filled with the right values. Because the labels declared external are not actually defined, the Assembler has no way of knowing the length, i.e., (byte, word or long) of the label. Therefore, a suffix must be used on each label in an **EXTERN** directive declaring its type; if the type is undefined, the current default label scope (set by BYTES, WORDS, LONGS directives) is assumed.
Example:	`EXTERN label.w, label1.b, label2.l`
See Also:	PUBLIC

A.18 #ELSE

Purpose:	Conditional ELSE.
Format:	`#ELSE`
Description:	This directive forces execution of the statements until the next **#ENDIF** if the last **#IF** statement was found false or disables execution of the statements until the next **#ENDIF** if the last **#IF** statement was found true. The **#ELSE** is optional in **#IF / #ENDIF** structures. In case of nested **#ELSE** statements, a **#ELSE** refers to the last #**IF**.

Example:	`#IF {1 eq 0}  ;`
	`  ; block A ... not assembled`
	`#ELSE`
	`  ; block B ... assembled`
	`#ENDIF`

See Also:	#IF, #ENDIF

A.19 #ENDIF

Purpose:	Conditional terminator.
Format:	`#ENDIF`
Description:	This is the non optional terminator of a **#IF** structure. If there is only one level of **#IF** nesting in force, then the statements after this directive will never be ignored, no matter what the result of the previous #IF was. In other words, the **#ENDIF** ends the capability of the previous #IF to suppress assembly. When used in a nested situation it does the same job, but if the last **#IF / #ENDIF** structure was in a block of source suppressed by a previous #IF still in force, the whole of the last **#IF / #ENDIF** structure will be ignored no matter what the result of the previous #IF was.

Example:	`#IF {count gt 0}`
	`...`
	`#ENDIF`

See Also:	#IF, #ELSE

A.20 FCS

Purpose:	Form constant string.
Format:	`FCS <"string"> |<bytes> [<"string"> |<bytes>]...`
Description:	This directive works the same as the common STRING directive, except that the last character in any string argument has bit 7 (e.g. MSB) forced high. Numeric arguments in the same list are left untouched.
Example:	`FCS "ALLO"  ; 41,4C,4C,CF` `STRING "ALLO"  ; 41,4C,4C,4F`
See Also:	STRING

A.21 .FORM

Purpose:	Set form length of the listing device.
Format:	`.FORM <exp>`
Description:	The Assembler paginates the listing (when selected) with a default of 66 lines per page. This directive changes the page length from the default. This directive does not generate assembly code or data.
Example:	`.FORM 72`
See Also:	TITLE, SUBTTL, %OUT, .LALL, .XALL, .SALL, .LIST,.NOLIST

A.22 GROUP

Purpose:	Name area of source code.
Format:	`GROUP <exp>`
Description:	All source code following a GROUP directive until the next GROUP directive or the end of the file - 'belongs' to the named group. Source code not included inside a group is allocated to a special group called 'Default'.
Example:	`GROUP mainloop`

A.23 #IF

Purpose:	Start conditional assembly.
Format:	`#IF <exp>`
Description:	Sometimes it is necessary to have different versions of a program or macro. This can be achieved by completely SEPARATE programs / macros, but this solution has the associated problem that changes to any part of the program common to all the versions requires all of them being changed, which can be tedious. Conditional assembly offers the solution of controlled 'switching off' assembly of the source code, depending on the value of the numeric expressions. The structure is known as '**IF/ELSE/ENDIF**': see the example for the format. The **#ELSE** statement is optional. If the expression resolves to 0 the expression is assumed to have a 'false' result: the source code between the false **#IF** and the next **#ENDIF** (or **#ELSE** if supplied) will not be assembled. If the **#ELSE** is supplied, the code following the **#ELSE** will be assembled only if the condition is false. **Conditionals may be nested up to 15 levels**: when nesting them, keep in mind that each **#IF** must have a **#ENDIF** at its level, and that **#ENDIFs** and **#ELSEs** refer to the last unterminated **#IF**.

Example:	`#IF {1 eq 1}`
	`%out true`
	`#FALSE`
	`%out false`
	`#ENDIF`

See Also:	#ENDIF, #ELSE, #IF1, #IF2

A.24 #IF1 Conditional

Purpose:	Conditional on being in pass #1.
Format:	`#IF1`
Description:	This directive works just like #IF except it has no argument and only evaluates itself as true if the Assembler is on its first pass through the source code. Can use #ELSE and requires #ENDIF.

Example:	`#IF1`
	`%OUT "Starting Assembly"`
	`#ENDIF`

See Also:	#IF2, #ELSE, #IF, #ENDIF

A.25 #IF2

Purpose:	Conditional on being in pass #2.
Format:	`#IF2`
Description:	This directive works just like #IF except it has no argument and evaluates itself as true only if the Assembler is on its second pass through the source code.

Example:	`#IF2`
	`%OUT "GONE through PASS-1 OK"`
	`#ENDIF`

See Also:	#IF1, #IF, #ENDIF, #ELSE

A.26 #IFB

Purpose:	Conditional on argument being blank.
Format:	`#IFB <arg>`
Description:	This directive works just like #IF except it doesn't evaluate its argument: it simply checks to see if it is empty or blank. Spaces count as blank.

Example:	`check  MACRO param1`
	`#IFB param1`
	`%OUT "No param1"`
	`#ELSE`
	`%OUT param1`
	`#ENDIF`
	`MEND`
	`...`
	`check ,`
	`check 5`

See Also:	#IF2, #ELSE, #IF, #END

A.27 #IFIDN

Purpose:	Conditional on arguments being identical.
Format:	`#IFIDN <arg-1> <arg-2>`
Description:	This directive works just like #IF except it compares two strings separated by a space. If identical, the result is true.

Example:	`check  MACRO param1`
	`#IFIDN param1 HELLO`
	`%OUT "Hello"`
	`#ELSE`
	`%OUT "No Hello"`
	`#ENDIF`
	`MEND`

See Also:	#IF2, #ELSE, #IF, #END

A.28 #IFDEF

Purpose:	Conditional on argument being defined.
Format:	`#IFDEF <exp>`
Description:	This directive works just like #IF except it tests for its argument being defined.

Example:	`check  MACRO param1`
	`#IFDEF param1`
	`%OUT "Arg is OK"`
	`#ELSE`
	`%OUT "Arg is undefined"`
	`#ENDIF`
	`MEND`

See Also:	#IF2, #ELSE, #IF, #END

A.29 #IFLAB

Purpose:	Conditional on argument being a label.
Format:	`#IFLAB <arg>`
Description:	This directive works just like #IF except it tests that its argument is a valid, predefined label.

Example:	`check MACRO param1`
	`#IFLAB param1`
	`%OUT "LABEL"`
	`#ENDIF`
	`MEND`

See Also:	#IF2, #ELSE, #IF, #END

A.30 #INCLUDE

Purpose:	Insert external source code file.
Format:	`#INCLUDE "<filename>"`
Description:	**INCLUDE** files are source code files in the same format as normal modules but with two differences: the first line is reserved for the processor name like any other source line, and they have no **END** directive. They contain **#DEFINE** and macro definitions that may be used by many different modules in your program. Instead of having each module declare its own set of #DEFINE and macro definitions, each module just includes the contents of the same #INCLUDE file. The Assembler processes the named INCLUDE file before returning to the line after the #INCLUDE directive in the source . Any alterations made to a macro are done once, in the include file, and the all modules referring to the changed entry are reassembled.

Example:	`st7/`
	`#include "defst7.h"`
	`...`
	`END`

A.31 INTEL

Purpose:	Force Intel-style radix specifier to be used during assembly
Format:	`INTEL`
Description:	The Intel style: **`0ABh`** Hexadecimal **`17o or 17q`** Octal **`100b`** Binary **`17`** Decimal (default) **`$`** Current program counter
Example:	`INTEL` `ld X,0FFFFh`
See Also:	MOTOROLA, TEXAS, ZILOG

A.32 INTERRUPT

Purpose:	Specifies for the debugger that a routine is an interrupt rather than a function.
Format:	`INTERRUPT <string>`
Description:	This directive is used with interrupt handlers and so aids the debugger in correctly searching the stack for return address of the interrupted function.

Example:	`PUBLIC trap_handler`
	`INTERRUPT trap_handler`
	`trap_handler IRET`

See Also:	NEAR

A.33 .LALL

Purpose:	List whole body of macro calls.
Format:	`.LALL`
Description:	This directive forces the complete listing of a macro expansion each time a macro is invoked. This is the default. This directive does not generate assembly code or data.
Example:	`.LALL`
See Also:	.XALL, .SALL

A.34 .LIST

Purpose:	Enable listing (default).
Format:	`.LIST`
Description:	This directive switches on the listing if a previous .**NOLIST** has disabled it. The **-'pa'** or **-'li'** options must also have been set from the command line to generate a listing. This directive, in conjunction with the directive **.NOLIST**, can be used to control the listing of macro definitions. This directive does not generate assembly code or data.
Example:	`.LIST`
See Also:	.NOLIST

A.35 #LOAD

Purpose:	Load named object file at link time.
Format:	`#LOAD "pathname\filename[.ext]"`
Description:	This directive leaves a message for the Linker to load the contents of the named file at the current position in the current segment. The file should be in **'straight binary'** format, i.e., a direct image of the bytes you want in the object code. It should not be in Motorola (.s19) or Intel (.hex) format.
Example:	`segment byte at 8000-C000 'EPROM1'` `#LOAD "table.bin"`
See Also:	

A.36 LOCAL

Purpose:	Define labels as local to macro.
Format:	`LOCAL <arg>`
Description:	A macro that generates loop code gives rise to an assembly problem since the loop label would be defined as many times as the macro is called. The LOCAL directive enables you to overcome this difficulty. Consider the following piece of code: waiter MACRO ads loop ld A,ads jrne loop MEND If this macro is called twice, you will be creating two labels called '**loop**'. The answer is to declare very early in the **MACRO** all labels created by the macro as **LOCAL**. This has the effect of replacing the actual name of a local label (here '**loop**') with **LOCXXXX** where **XXXX** starts from 0 and increments each time a local label is used. This provides each occurrence of the labels created inside the macro with a unique identity.

Example:	`waiter MACRO ads`
	`LOCAL loop`
	`loop led Aids`
	`drone loop`
	`MEND`

See Also:	MACRO, MEND

A.37 LONG

Purpose:	Define long word in object code.
Format:	`LONG <exp>[,<exp>...]`
Description:	This directive forces the long word(s) in its argument list into the object code at the current address. The arguments may be composed of complex expressions, which may include external labels. If the argument is an expression with a value greater than FFFFFFFF, then the 32 bits of the expression are used and no errors are generated. **LONG** sends long words with the least significant byte first. Generally used to define data tables. Synonymous with **DC.L**, except that **LONG** sends the low-byte first.
Example:	`DC.L 1,$12345678 ; 0000,0001,1234,5678` `LONG 1,$12345678 ; 0100,0000,7856,3421`
See Also:	DC.B, DC.L, DC.W, BYTE, STRING, WORD

A.38 LONGS

Purpose:	Default new label length long.
Format:	`LONGS`
Description:	When a label is defined, four SEPARATE attributes are defined: scope (internally or externally defined), value (actual numerical value), relativity (**absolute** or **relative**), and lastly, length (BYTE, WORD or LONG). All attributes except length are defined explicitly before or at the end of the definition. Force a label to a length with a dot suffix, e.g. 'label.b' for byte. You may also define a default state for label length: labels are created to this length unless otherwise forced with a suffix. The default is set to WORD at the start of the assembly, but may be changed by BYTES, WORDS or LONGS to the appropriate length.
Example:	`LONGS` `lab1 EQU 5 ; long length for lab1`
See Also:	BYTES, WORDS

A.39 MACRO

Purpose:	Define macro template.
Format:	`<macro> MACRO [param-1][,param-2]...`
Description:	This directive defines a macro template that can be invoked later in the program. The label field holds the name of the macro: this name is used to invoke the rest of the macro whenever it is found in the opcode field. The arguments are dummy names for parameters that will be passed to the macro when it is used: these dummy names will be replaced by the actual calling line's arguments.

Example:	`cmp16 MACRO first,second,result`
	`LOCAL trylow`
	`ld A,first`
	`add A,second`
	`cp A,#0`
	`jreq trylow`
	`cpl A`
	`trylow ld result,A`
	`MEND`

See Also:	MEND, .LALL, .SALL, .XAL

A.40 MEND

Purpose: End of macro definition.

Format: `MEND`

Description: End of macro definition.

Example:

```
cmp16   MACRO first,second,result
        LOCAL trylow
        ld A,first
        add A,second
        cp A,#0
        jreq trylow
        cpl A
trylow  ld result,A
        MEND
```

See Also: MACRO

A.41 MOTOROLA

Purpose: Force Motorola-style radix specifier.

Format: `MOTOROLA`

Description: The Motorola style:

$AB	Hexadecimal
~17	Octal
%100	Binary
17	Decimal (default)
*****	Current program counter

This directive forces the Motorola format to be required during the assembly. The default format is **MOTOROLA**.

Example:

```
        MOTOROLA
        ld X,$FFFF
```

See Also: INTEL, TEXAS, ZILOG

A.42 NEAR

Purpose:	Specifies to debuggers that the return address in the stack for functions using this directive is written over two bytes.
Format:	`NEAR <"string">`
Description:	This directive is used with functions called by CALL or CALLR, whose return stack address spans two bytes. Every function called by CALL or CALLR must be classified as NEAR.

Example:	`PUBLIC func`
	`NEAR func`
	`func ret`

See Also:	INTERRUPT

A.43 .NOCHANGE

Purpose:	List original #define strings.
Format:	`.NOCHANGE`
Description:	Strings named in the first argument of a #DEFINE directive will be changed to the second argument of the #DEFINE: the default is that the changed strings will be listed. If you want the original source code to be listed instead, place a .NOCHANGE directive near the start of your source code. This directive does not generate assembly code or data.
Example:	`.NOCHANGE`
See Also:	#DEFINE

A.44 .OFFSETST7

Purpose:	Specify the first MSCI address in ST7 memory. Warning: This address is **byte-based** rather than word-based.
Format:	`.OFFSETST7<value>`
Description:	Allows the user to indicate the very first address of MSCI memory in the overall ST7 address space. The address `<value>` can be in decimal or hexadecimal. Hexadecimal values are proceeded by $. There must not be any other signs on the directive line, including comments, spaces or tabulations.
Example:	`.OFFSETST7<$1620>`
See Also:	

A.45 .ORG

Purpose:	Indicates that the code is to placed in a specific area of the MSCI memory space. Warning: This address is **byte-based** rather than word-based.
Format:	`.ORG<name><value>`
Description:	Allows the user to indicate where code is to be place within the MSCI memory space. The address `<value>` can be in decimal or hexadecimal. Hexadecimal values are proceeded by $. There must not be any other signs on the directive line, including comments, spaces or tabulations.
Example:	`.OFFSETST7<$1620>`
See Also:	

A.46 %OUT

Purpose:	Output string to the console.
Format:	`%OUT string`
Description:	This directive prints its argument (which does not need to be enclosed in quotes) to the console. This directive does not generate assembly code or data.
Example:	`%OUT hello!`
See Also:	

A.47 .PAGE

Purpose:	Perform a form feed.
Format:	`.PAGE`
Description:	Forces a new page listing. This directive does not generate assembly code or data.
Example:	`.PAGE`
See Also:	

A.48 PUBLIC

Purpose: Make labels public.

Format: `PUBLIC <arg>`

Description: This directive marks out given labels defined during an assembly as **'PUBLIC'**, i.e., accessible by other modules. This directive is related to **EXTERN**; if one module wants to use a label defined in another, then the other module must have that label declared **PUBLIC**.

A label may also be declared PUBLIC as its definition by preceding the label name with a dot; it needn't be declared in a PUBLIC directive then.

Example:

```
module1.asm
        EXTERN print.w, print1.w
        ...
        call print
        ...
        jp print1
module2.asm
        PUBLIC print
        print nop
        .print1 nop
```

See Also: EXTERN

A.49 REPEAT

Purpose: Assembly-time loop initiator.

Format: `REPEAT`

Description: Used together with UNTIL to make assembly-time loops; it is useful for making tables etc. This directive should not be used within macros.

Example:

```
        REPEAT
```

See Also: CEQU, UNTIL

A.50 SEGMENT

Purpose:	Start of new segment.
Format:	`[<name>] SEGMENT <align> <combine> '<class>' [cod]`
Description:	The SEGMENT directive is very important: every module in your program will need at least one. The **<name>** field may be up to 11 characters in length, and may include underscores. The <align> field is one of the following:
Align Type:	**byte** – no alignment; can start on any byte boundaries **word** – aligned to next word boundaries if necessary, i.e., 8001=8002 **para** – aligned to the next paragraph (=16 bytes) boundary, i.e., 8001=8010 **64** – aligned to the next 64-byte boundary, i.e., 8001=8040 **128** – aligned to the next 128-byte boundary, i.e., 8001=8080 **page** – aligned to the next page (=256 bytes) boundary, i.e., 8001=8100 **long** – aligned to the next long-word(=4 bytes) boundary, i.e., 8001=8004 **1K** – aligned to next 1K boundaries, i.e., 8001=8400 **4K** – aligned to next 4K boundaries, i.e., 8001=9000
Combine: at	**X[-Y]** – Introduces new class that starts from X and goes through to address Y. Address Y is optional. **<none>** – Tack this code on the end of the last segment of this class. **common** – Put the segment at the same address than other common segments that have the same name, and note the longest length segment. The optional **[cod]** suffix is a number from 0 to 9 - it decides into which. COD file the Linker sends the contents of this class. 0 is the default and is chosen if the suffix is left off. A suffix of 1-9 will cause the Linker to open the [cod] suffix, and send the contents of this class into the cod file instead of the default. This allows bank switching to be supported directly at link level- different code areas at the same address can be separated out into different .cod files.
See Also:	For more information, see *Section 4.4* on page 28.

A.51 .SETDP

Purpose:	Set base address for direct page.
Format:	`.SETDP <base address>`
Description:	If you have used an ST7 processor, you'll be aware of its 'zero-page' or 'direct' addressing modes. These use addresses in the range 00..FF in shorter, faster instructions than the more general 0000..FFFF versions. Other processors use the same scheme, but with a twist: you can choose the 'base page' where the direct mode does not have to be in range 0000 00FF but can be from nn00..nnFF where nn00 is the 'base page', loaded into a register at run-time. Because the Assembler cannot track what's in the base page register at run-time, you need to fill it in about the current 'base page' with the .SETDP directive. At the start of the assembly, SETDP defaults to 0000.
Example:	`.SETDP $400` `ld A,$401 ; direct mode chosen`
See Also:	

A.52 SKIP

Purpose:	Inserts given number of bytes with an initialization value.
Format:	`SKIP <number of bytes>,<value to fill>`
Description:	This directive leaves a message for the Linker that you want X number of Y bytes to be inserted into the object code at this point. Both the arguments must be absolute values rather than external or relative values.
Example:	`SKIP 100,$FF ; insert 100 bytes all $FF`
See Also:	

A.53 STRING

Purpose:	Define a byte-level string.
Format:	`STRING <exp or "string">,[,<exp or "string">...]`
Description:	This directive forces the byte(s) in its argument list into the object code at the current address. The arguments may be composed of complex expressions, which may even include external labels. If the argument was an expression and had a value greater than 255 the lower 8 bits of the expression are used and no errors are generated. String argument(s) must be surrounded by double-quotes: these are translated into ASCII and processed byte by byte. It's generally used for defining data tables. Synonymous **BYTE** and **DC.B.**

Example:	`STRING 1,2,3 ; generates 01,02,03`
	`STRING "HELLO" ; generates 48,45,4C,4C,4F`
	`STRING "HELLO",0 ; generates 48,45,4C,4C,4F,00`

See Also:	DC.B, BYTE, WORD, LONG, DC.W, DC.L, FCS

A.54 SUBTTL

Purpose:	Define a subtitle for listing heading.
Format:	`SUBTTL "<Subtitle string>"`
Description:	This directive is related to the **TITLE** directive: its argument is used as a subtitle at the beginning of each page on a listing. We recommend that individual subtitles are generated for each module in a program, while the **TITLE** is defined once in the include file called by all the modules. This directive does not generate assembly code or data.
Example:	`SUBTTL "A/D control routines"`
See Also:	TITLE

A.55 TEXAS

Purpose:	Texas Instruments-style radix specifier.
Format:	`TEXAS`
Description:	The Motorola style: **`>AB`** Hexadecimal **`~17`** Octal **`?100`** Binary **`17`** Decimal (default) **`$`** Current program counter This directive forces the Texas Instruments format to be required during the assembly.
Example:	`TEXAS` `ld X,>FFFF`
See Also:	INTEL, MOTOROLA, ZILOG

A.56 TITLE

Purpose:	Define main title for listing.
Format:	`TITLE "<Title string>"`
Description:	The first fifty-nine characters of the argument (which must be enclosed in double-quotes) will be included on the first line of each page in a listing as the main title for the listing. We suggest you set the title in the include file called by each module in your program, and give each module a separate subtitle (see SUBTTL section). This directive does not generate assembly code or data.
Example:	`TITLE "ST7 controller program"`
See Also:	SUBTTL

A.57 UNTIL

Purpose:	Assembly time loop terminator.
Format:	`UNTIL <exp>`
Description:	Related to REPEAT directive: if the expression in the argument resolves to a non zero value then the Assembler returns to the line following the last REPEAT directive. This directive cannot be used inside macros.

Example:	`val  CEQU 0`
	`  REPEAT`
	`  DC.L {10 mult val}`
	`val  CEQU {val+1}`
	`  UNTIL {val eq 50}`

See Also:	CEQU, REPEAT

A.58 WORD

Purpose:	Define word in object code.
Format:	`WORD <exp>[, <exp>...]`
Description:	This directive forces the word(s) in its argument list into the object code at the current address. The arguments may be composed of complex expressions that may even include external labels. If the argument was an expression and had a value greater than FFFF then the lower 16 bits of the expression are used and no errors are generated. WORD sends the words with the least significant byte first. It's generally used for defining data tables. Synonymous with DC.W.
Example:	`WORD 1,2,3,4,$1234  ;0001,0002,0003,0004,1234`
See Also:	DC.B, BYTE, STRING, DC.W, LONG, DC.L

A.59 WORDS

Purpose:	Default new label length word.
Format:	`WORDS`
Description:	When a label is defined, four SEPARATE attributes are defined with its scope (internal or external defined) value (actual numerical value of the label) relativity (the label is ABSOLUTE or RELATIVE), and lastly length (BYTE, WORD, or LONG). All attributes except length are defined explicitly before or at the definition. Force the label to a certain length with a dot suffix, e.g. '.b' for byte length. You may also define a default state for label length: the label is created to this length unless otherwise forced with a suffix. The default is set to WORD at the start of the assembly, but may be CHANGED by **BYTES, WORDS** or **LONGS** to the appropriate length.
Example:	`WORDS` `lab1 EQU 5 ; word length for lab1`
See Also:	BYTES, WORDS

A.60 .XALL

Purpose:	List only code producing macro lines.
Format:	`.XALL`
Description:	This directive forces a reduced listing of a macro expansion each time a macro is invoked. Only those lines of the macro that generated object code are listed. This instruction itself is not listed. This directive does not generate assembly code or data.
Example:	`.XALL`
See Also:	.LALL,.SALL

A.61 ZILOG

Purpose:	Force Zilog-style radix specifiers.
Format:	`ZILOG`
Description:	The Motorola style: **`%AB`** Hexadecimal **`%(8)17`** Octal **`%(2)100`** Binary **`17`** Decimal (default) **`$`** Current program counter This directive forces the Zilog format to be required during the assembly.
Example:	`ZILOG` `ld X,%FFFF`
See Also:	INTEL, MOTOROLA, TEXAS

Appendix B Error Messages

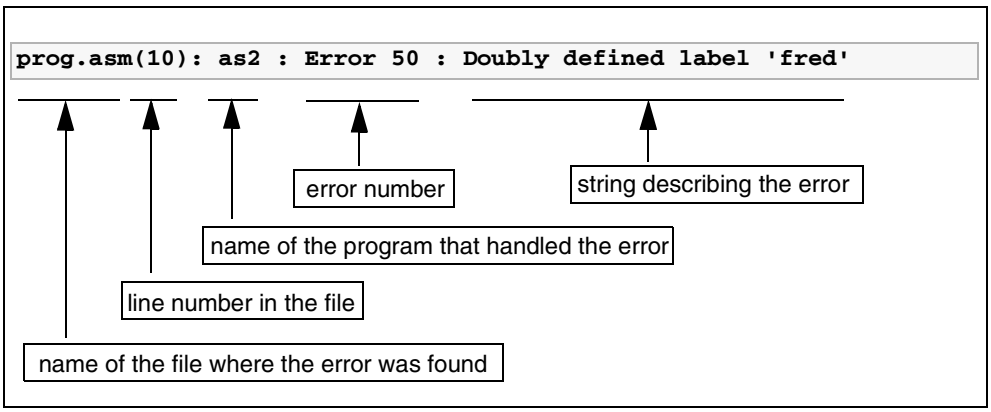
B.1 Format of error messages

There are two classes of errors trapped by the Assembler: **fatal** and **recoverable**. A fatal error stops the assembly when they are detected, returning you to the caller with a message and error number describing the problem.

Format of the error messages is:

```
file.asm(line): as<pass> : Error <errno> : <message> '<text>'
```

Figure 3: Example of an error message



*Note: The name of the program that handled the error (third field), can be **as1** or **as2** depending on the pass in progress when the error was found.*

The error number (fourth field) can be used as an index to find a more complete description of the error in the next section (fatal errors read 'FATAL nn', instead of 'ERROR nn').

B.2 File CBE.ERR

Fatal and recoverable errors are copied into the file CBE.ERR as they occur. CBE can use this error file to give automatic error-finding. Most Linker errors are also copied into CBE.ERR.

B.3 Assembler and linker errors

For a complete listing of error messages and there meanings, refer to the *ST7 Assembler Linker User Manual, Appendix B*.

Appendix C Revision History

Date	Revision	Description of changes
30- Jun-2005	1	• Initial release

Appendix D Product Support

If you experience any problems with this product, or contact the distributor or the STMicroelectronics sales office where you purchased the product. Phone numbers for major sales regions are provided in the *Contact List*, below.

In addition, at our Internet site **www.st.com/mcu**, you will find a complete listing of ST sales offices and distributors, as well as documentation, software downloads and user discussion groups to help you answer questions and stay up to date with our latest product developments.

D.1 Software updates

5. All our latest software and related documentation are available for download from the ST Internet site, **www.st.com/mcu**.

Contact List

North America

Canada and East Coast

STMicroelectronics
Lexington Corporate Center
10 Maguire Road, Building 1, 3rd floor
Lexington, MA 02421
Phone: (781) 402-2650

Mid West

STMicroelectronics
1300 East Woodfield Road, Suite 410
Schaumburg, IL 60173
Phone: (847) 585-3000

West coast

STMicroelectronics, Inc.
1060 E. Brokaw Road
San Jose, CA 95131
Phone: (408) 452-8585

Note: For American and Canadian customers seeking technical support the US/Canada is split in 3 territories. According to your area, contact the appropriate sales office from the list above and ask to be transferred to an 8-bit microcontroller Field Applications Engineer.

Europe

France +33 (0)1 47 40 75 75
Germany +49 89 46 00 60
U.K. +44 162 889 0800

Asia/Pacific Region

Japan +81 3 3280 4120
Hong-Kong +85 2 2861 5700
Shanghai +86 21 52574828
Sydney +61 2 9580 3811
Taipei +88 6 2 2378 8088

Index

E

F

I

L

M

N

O

P

R

S

T