UM0144
User Manual

ST Assembler-Linker

Introduction

This manual describes using the ST Assembler-Linker to develop applications for the ST7 and STM8 microcontrollers. The assembly tools described in this book form a development system that assembles, links and formats your source code.

Purpose and scope

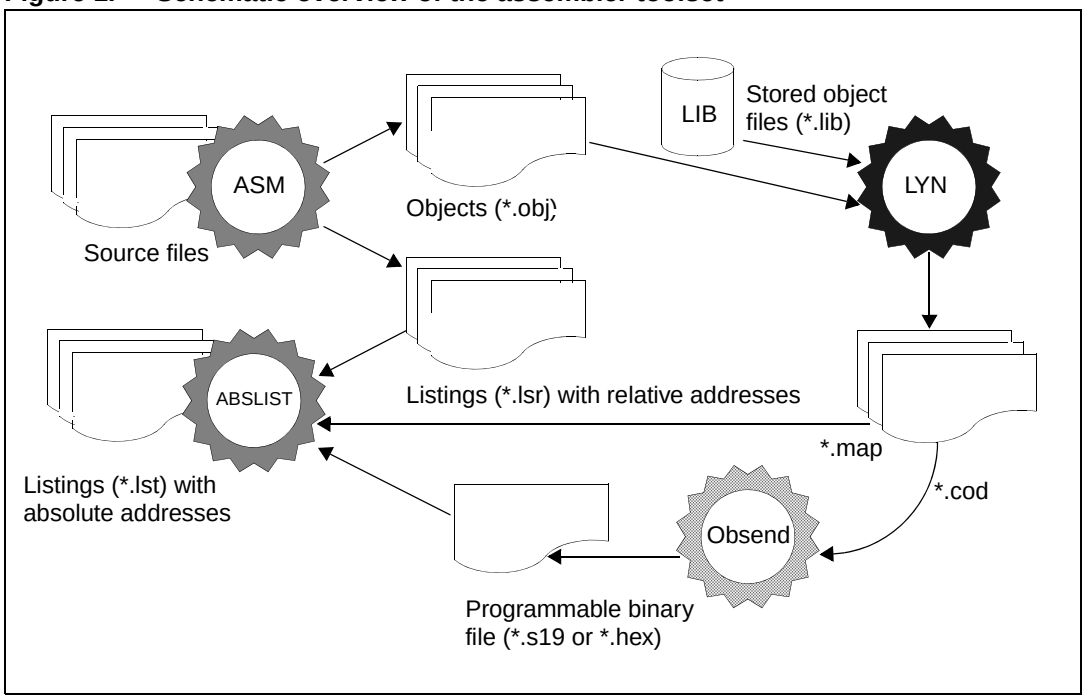
This manual provides information about producing an application executable for the ST7 and STM8 microcontrollers from your application source code in assembler language. It includes:

- An overview of assembly language for the ST7 and STM8 microcontrollers
- Instructions for running the ST Assembler-Linker
- Descriptions of the assembler output

For information on related subjects refer to the following documentation:

- ST7xxxx Datasheet – full description of the ST7 xxxxmicrocontroller
- STM8xxxx Datasheet – full description of the STM8xxxx microcontroller
- ST7 Programming Manual – a complete reference to the ST7 assembly language
- STM8 Programming Manual – a complete reference to the STM8 assembly language

Figure 1. Schematic overview of the assembler toolset



Contents

List of tables

List of figures

1 Introduction

1.1 ST7 and STM8 incompatibilities

The new ST7/STM8 assembler development toolchain supports both the new STM8 core and the old ST7 core. By placing a trigger (`st7/` or `stm8/`) in the first line of your code, you tell the assembler which set of rules to apply.

The STM8 assembler is not compatible with the ST7 assembler.

STM8 assembler (16-bit) features that are not compatible with the ST7 assembler (8-bit):

1. X and Y are 16 bits wide (ST7 is 8 bits):
   `ld A,X` has been replaced by `ld A,XL`
   `ld X,A` has been replaced by `ld XL,A`
   `ld A,Y` has been replaced by `ld A,YL`
   `ld Y,A` has been replaced by `ld YL,A`
2. Stack pointer (SP) is 16 bits wide (ST7 is 8 bits wide):
   `ld A,S` and `ld S,A` instructions have been removed
   `ld X,S` has been replaced by `ldw X,SP`
   `ld S,X` has been replaced by `ldw SP,X`
   `ld Y,S` has been replaced by `ldw Y,SP`
   `ld S,Y` has been replaced by `ldw SP,Y`
3. more generally
   `ld` is for an 8-bit transfer, for example: `ld A,$5000`
   `ldw` is for 16-bit transfer, for example: `ldw X,$5000` (instead of `ld X,$5000`)
4. `RSP` instruction has been removed
5. Some addressing modes have been removed, for example:
   - short pointer to short data [pointer.b], for example:
   ```
   ld A,[$10.b]
   btjf [$11.b],#3,skip
   ```
   - short pointer to short data X or Y indexed ([pointer.b],X) or ([pointer.b],Y), for example:
   ```
   ld A,([$10.b],X)
   ld ([$12.b],Y),A
   ```
   - short pointer to short data relative [pointer.b], for example:
   ```
   jreq [$13.b]
   callr [$39.b]
   ```
6. Short bit operations have been replaced by long bit operations, for example:
   `btjf $1011,#2,jump` (instead of `btjf $11,#2,jump`)
   `bset $1000,#1` (instead of `bset $00,#1`)
7. `.h` and `.l` suffixes are not supported, for example:
   `ld A,#mem.h` can be replaced by `ld A,#{high mem}`
   `ld A,#mem.l` can be replaced by `ld A,#{low mem}`

Generally, the instruction sets are similar, with the following notable differences:

- The STM8 instruction set supports several new addressing modes.
  - The Stack Pointer (SP) can be used as an index.
  - Long pointers have been added.
  - There is a new 3-byte addressing mode called extended.
  - Altogether there are 6 new addressing modes:
    short offset SP indexed,
    extended direct,
    extended offset X or Y indexed,
    long pointer to long data,
    long pointer to long data X indexed,
    long pointer to 24-bit data X or Y indexed.
- Several new instructions have been added.
- The STM8 instruction set allows for longer instructions which may span 5 bytes, instead of 4 for the ST7.

1.2 Tools

The ST Assembler-Linker includes the following tools:

- **Assembler (ASM)**: translates your source code (**`.ASM`**) written in assembly language, into object code (**`.OBJ`**) specific to the target machine and a listing file with relative addresses(**`.LSR`**).
- **Linker (LYN)**: processes the object files (**`.OBJ`**) produced by the assembler, resolves all cross-references between object files and locates all the modules in memory. The resulting code is output in an object code file (**`.COD`**).
- **Converter (OBSEND)**: translates the object code file to produce the final executable in a format that you specify (Motorola S-record, Intel Hex).
- **List file postprocessor (ABSLIST)**: patches the list file generated by the assembler to produce a new list file with absolute addresses (**`.LST`**).
- **Librarian (LIB)**: The librarian enables you to store frequently used subroutines in one location for use with any number of ST microcontroller applications.

*Note: The utility file **`asli.bat`** automatically runs ASM, LYN, OBSEND and ABSLIST one after the other for you. Use this batch file only if you have only one assembly source file .ASM.*

1.3 Host PC system requirements

Please see the release notes to ensure you have the most up-to-date information.

1.4 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 web site: www.st.com.

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. We'll be glad to help you.

1.5 Conventions

The following conventions are used in this document:

- **Bold text** highlights key terms and phrases, and is used when referring to names of dialog boxes and windows, as well as tabs and entry fields within windows or dialog boxes.
- ***Bold italic*** text denotes menu commands (or sequence of commands), options, buttons or checkboxes which you must click with your mouse to perform an action.
- The **>** symbol is used in a sequence of commands to mean "then". For example, to open an application in Windows, we would write: Click ***Start>Programs>ST Toolset>***.
- `Courier` font designates file names, programming commands, path names and any text or commands you must type.
- *Italicized* type is used for value substitution. *Italic* type indicates categories of items for which you must substitute the appropriate values, such as arguments, or hypothetical filenames. For example, if the text was demonstrating a hypothetical command line to compile and generate debugging information for any file, it might appear as:

  `cxst7 +mods +debug` *file*`.c`
- Items enclosed in [brackets] are optional. For example, `[`*`options`*`]` means that zero or more options may be specified because options appears in brackets. Conversely, the line: *`options`* means that one or more options must be specified because options is not enclosed by brackets. As another example, the line:

  ```
  file1.[o|st7]
  ```

  means that one file with the extension `.o` or `.st7` may be specified, and the line:

  ```
  file1 [file2...]
  ```

  means that additional files may be specified.

Blue italicized text indicates a cross-reference—you can link directly to the reference by clicking on it while viewing with Acrobat Reader.

2 Getting started

Installing the ST Assembler-Linker

The ST Assembler-Linker is delivered as part of the STVD toolset. A free installation package is available at **www.st.com**. To install it:

- either select ***ST7/STM8>ST toolset*** from the main menu of the Microcontroller Development Tools CD-ROM,
- or run the installation executable that you have downloaded from the internet.

Note: See the release notes for more guidance on installing the software components.

After installation, the installation directory should contain the files listed in *Table 1*.

Table 1. Description of installed files

ASM.EXE	ST assembler
LYN.EXE	ST linker
OBSEND.EXE	Output file formatter
ABSLIST.EXE	List file post processor
LIB.EXE	Librarian
ST7.TAB	ST7 description file
STM8.TAB	STM8 description file
ASLI.BAT	Batch file ASM+LYN+OBSEND+ABSLIST
ASM_LNK_RELEASE_NOTES.PDF	Release notes

Up-to-date release notes are provided in PDF format. An additional file contains demonstration examples.

3 ST7 and STM8 addressing modes

3.1 Overview of ST7 and STM8 addressing modes

The ST7/STM8 assembler instruction set incorporates the following addressing modes:

Table 2. ST7 and STM8 addressing modes

Addressing mode	Example
Inherent	`nop`
Immediate	`ld A,#$F5`
Direct (short address)	`ld A,$F5`
Direct (long address)	`ld A,$F5C2`
X or Y indexed (no offset)	`ld A,(X)`
X or Y indexed (short offset)	`ld A,($F5,X)`
X or Y indexed (long offset)	`ld A,($F5C2,X)`
Short pointer indirect (long pointed data)	`ld A,[$F5.w]`
Short pointer indirect (long pointed data) X or Y indexed	`ld A,([$F5.w],X)`
Direct relative (short offset)	`jrt $F5`

Table 3. ST7 only addressing modes

Addressing mode	Example
Short pointer indirect (short pointed data)	`ld A,[$F5]`
Short pointer indirect (short pointed data) X or Y indexed	`ld A,([$F5],X)`
Short pointer indirect relative (short pointed data)	`jrt [$F5]`
Short bit operation	`bset $10, #5`

Table 4. STM8 addressing modes

Addressing mode	Example
Direct (extended address)	`callf $F5C2A0`
SP indexed (short offset)	`ld A,($F5,SP)`
X or Y indexed (extended offset)	`ldf A,($F5C2A0,X)`
Long pointer indirect (long pointed data)	`ld A,[$F5C2.w]`
Long pointer indirect (long pointed data) X indexed	`ld A,([$F5C2.w],X)`
Long pointer indirect (extended pointed data) X or Y indexed	`ldf A,([$F5C2.e],X)`
Long bit operation	`bset $1000, #1`

All the ST7 and STM8 addressing modes are described in full detail, with specific examples, in the relevant programming manual, which can be downloaded from the internet at

www.st.com. This chapter only gives a brief explanation of the main addressing mode types.

3.2 General instruction syntax

The ST7 and STM8 instruction sets provide a single source-coding model regardless of which components are operands.

- For the ST7 the operands may be:
  - the accumulator (A),
  - an 8-bit index register (X or Y)
  - an 8-bit stack pointer (S)
  - the condition code register (CC), or a memory location.
- For the STM8 the operands may be:
  - the accumulator (A),
  - a 16-bit index register (X or Y)
  - XH,XL (where XH is the high byte, and XL is the low byte)
  - YH,YL (where YH is the high byte, and YL is the low byte)
  - a 16-bit stack pointer (SP)
  - the condition code register (CC), or a memory location.

For example, a single instruction, **`ld`**, originates register to register transfers as well as memory to accumulator data movements.

Two-operand instructions are coded with the destination operand in the first position. For example,

```
lab01   ld A,memory ;  load accumulator A with memory contents
lab02   ld memory,A ;  load memory location with A contents
        ld X,A      ;  load X with accumulator contents (ST7 only)
        ld XL,A     ;  load XL with accumulator contents (STM8 only)
```

3.3 Short and long addressing modes

The ST7 has two addressing modes that differ in memory address size (one byte for short mode and two bytes for long mode).

For the STM8, in addition to long and short modes, there is also an extended addressing mode (three bytes).

Because of these different addressing modes, the target address range of the operands depends upon the addressing mode chosen:

- `0-$FF` for short addressing mode
- `$100-$FFFF` for long addressing mode
- `$10000-$FFFFFF` extended addressing mode (STM8 only)

Some instructions accept both long and short addressing modes, while others only accept one or the other. For example:

```
lab10   add A,memory   ;  accepts both types of addressing modes
lab11   inc memory     ;  ST7 instruction accepts only short
```

```
                        addressing mode, while STM8 instruction
                        accepts both modes
        push memory     ; STM8 accepts only long addressing mode, push
                        memory does not exist for ST7
```

For ST7 instructions supporting both short and long formats, when external symbols are referenced, long mode is chosen by the assembler.

For example:

```
        EXTERN symb3;
symb1   equ $10;
...
        ld A,symb1; short mode
        ld A,symb3; long mode chosen
```

STM8 instructions using the extended addressing mode always have an F suffix. The following instructions use the extended addressing mode:

```
callf   $10000
jpf     $20000
ldf     A,($30000,X)
retf ; permits you to return to the previous function in the stack
in subroutines that are called by CALLF
```

3.4 Inherent addressing mode

This concept is hardware-oriented, meaning that instruction operands are coded inside the operation code. At source code level, operands are written explicitly.

For example:

```
lab06   push  A   ; put accumulator A onto the stack
lab07   mul   X,A ; multiply X by A
        ldw SP,X  ; load X to the stack pointer
```

3.5 Immediate operands

Immediate operands permit you to input a specific value for use with an instruction. They are signaled by the use of a sharp sign (#) before the value. The range for an 8-bit immediate operand is from 0 to 255.

For example:

```
lab08   ld    A,#1            ; load A with immediate value 1
lab09   bset  memory,#3       ; set bit #3 in memory location
        btjt  memory,#3,label ; test bit #3 of memory and jump if
                                true (set)
```

3.6 Direct and indirect modes

A **direct addressing mode** means that the data byte(s) 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(s) required to do the operation is found by its memory address which is located in memory (pointer).

The pointer address follows the op-code. A short pointer is one byte long. A long pointer is two bytes long.

This last group consists of memory indirect variants:

- Short pointer to short data,
  for ST7 only [shortpointer .b]
- Short pointer to long data [shortpointer .w]
- Short pointer to short data X or Y indexed,
  for ST7 only ([shortpointer .b],X) ([shortpointer .b],Y)
- Short pointer to long data X or Y indexed ([shortpointer .w],X) ([shortpointer .w],Y)
- For STM8 devices only:
  - long pointer to long data [longpointer .w]
  - long pointer to long data X indexed ([longpointer .w], X)
  - long pointer to extended data X or Y indexed ([longpointer .e],X)([longpointer .e],Y)
- Pointer addresses must always be in:
  - page 0 (its address must be less than $100) for the ST7
  - section 0 (its address must be less than $10000) for the STM8

  Examples:

```
      ld A,[80]           short pointer to short (ST7) or long (ST8) data
      ld A,[80.b]         short pointer to short data (ST7 only)
      ld A,[80.w]         short pointer to long data
      ld A,[$1000.w]      long pointer to long data (STM8 only)
      ldf A,([$1000.e],X) long pointer to 24-bit data (STM8 only)
lab12 equ 80
      ld A,([lab12],X)    short pointer to short (ST7) or long (ST8) data X-indexed
      ld A,([lab12.b],X)  short pointer to short data X-indexed (ST7 only)
      ld A,([lab12.w],Y)  short pointer to long data Y-indexed
```

- To distinguish between short and long indirect addressing mode, the suffix **.w** indicates that you want to work in long indirect mode (this is also true for indexed addressing mode).
  - Short indirect means that pointed data are short (one byte long)
  - Long indirect means pointed data are long (two bytes long)
- Implicitly, if nothing is specified,
  - for the ST7, short indirect addressing mode is assumed, you can also use **.b** to specify short indirect addressing mode (as with the indexed addressing mode). Use .w to specify long indirect addressing mode.
  - for the STM8, long indirect addressing is assumed, you could use **.w** but it is not necessary. With the STM8 **ldf** instruction, you must use **.e** to specify extended indirect addressing mode.

3.7 Indexed modes

The ST7 supports the following types of indexed mode:

- indexed without offset,
- indexed with an 8-bit unsigned offset (range [0:255]),
- indexed with a 16-bit offset.

In addition to these modes, the STM8 also supports the following indexed mode:

- indexed with a 24-bit offset.

The source code syntax is:

- `(X)` or `(Y)` for no-offset indexing.
- `(offset,X)` or `(offset,Y)` for indexed with offset.

Some instructions (such as **`ld A`** or **`add`**) support the first three types of indexed mode. Some ST7 instructions (such as **`inc`**) only support the first two types (that is, indexed without offset and indexed with 8-bit unsigned offset).

The STM8 instructions (such as `inc`) support the first three types.

Only the STM8 instruction, `ldf`, supports the “indexed with 24-bit offset” addressing mode.

Examples:

```
ld A,(X)            ; no-offset mode
ld A,(0,X)          ; 8-bit offset mode
ld A,(127,X)        ; 8-bit offset mode
ld A,(259,X)        ; 16-bit offset mode
ldf A,($FFF00, X)   ; 24-bit offset mode (STM8 only)
ld A, ($F5, SP)     ; SP indexed mode, 8-bit offset short(STM8 only)
```

3.8 Relative mode

This addressing mode is used to modify the program counter (PC) register value by adding an 8-bit signed offset to it (in the range -128 to +127). The relative addressing mode is made up of two sub-modes:

- **relative (direct)** where the offset follows the op-code. This is used by the instructions **`JRxx, CALLR,`** and **`BTJx`**.
- **relative (indirect)** where the offset is defined in memory, this address follows the op-code (ST7 only).

The target label is specified at source code level (the assembler computes the displacement).

3.9 High, low addressing modes

In some instances, it may be necessary to access the highest part of an address (8 highest bits) or the lowest part of an address (8 lowest bits) as well.

For this feature in the ST7, the syntax is the following: **<expression>**, where **<expression>** is `symbol.H` (highest part), or `symbol.L` (lowest part). Examples:

```
lab12  equ $0012
```

```
        nop
        ld A,#lab12.h; load A with $00
        ld A,#lab12.l; load A with $12
```

In the STM8, symbols `.H` and `.L` are not available. Use low and high primitives instead for example:

```
lab1    equ $112233
        ld A,#{low{seg lab1}}; load A with $11
        ld A,#{high lab1}     ; load A with $22
        ld A,#{low lab1}      ; load A with $33
```

4 ST assembler

4.1 Overview

The ST assembler program is a cross-assembler, meaning that it produces code for a target machine (an ST7 or STM8 microprocessor) which is different from the host machine.

The assembler turns the source code files into re-locatable object modules ready for linking.

During the process, it checks for many 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 *Appendix B: Error messages* on page 78.

To produce code ready for execution, you must run the assembler (**ASM**), the linker (**LYN**), and the object code formatter (**OBSEND**).

4.2 Source files

Source program code is written in the ST7 or STM8 assembler language and is saved in an ASCII text file named **source file**. A source file has the extension **`.asm`**. It is made up of lines, each of which is terminated by a new line character.

For a complete reference of the ST7 or STM8 assembler language, refer to the relevant programming manual.

4.3 Assembler source code format

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

Use this line to specify the directory location of the *.tab file. If the directory is not specified, by default the Assembler searches first in the current directory, then in the directory where the Assembler's executable is located.

The '.tab' suffix may be left out, as the assembler only looks for this file type.

The first line of your source code might look like:

`st7\` or `c:\sttools\asm\st7\` (to use the ST7 processor)

`stm8\` or `c:\sttools\asm\stm8\` (to use the STM8 processor)

If the file `st7.tab` (or `stm8.tab`) cannot 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]
```

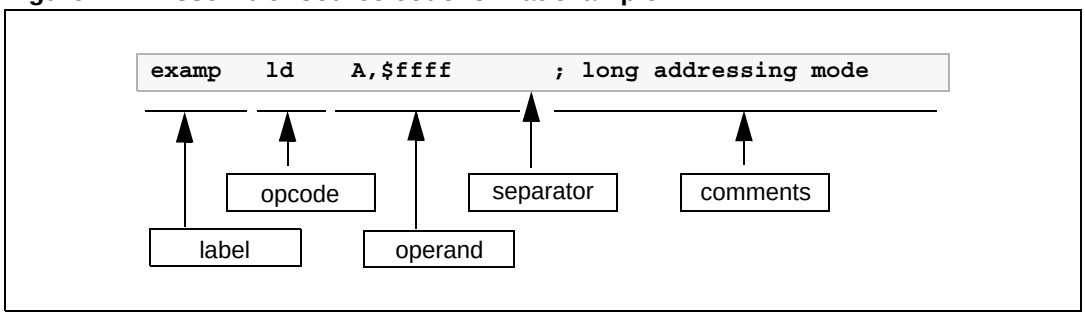
where `<space>` refers to either a SPACE (**`$20`**) or a TAB (**`$09`**) character.

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.

For example:

Figure 2. Assembler source code format example



The next sections describe the main components of a source code file.

4.3.1 Label structure

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. Note that upper and lower case are treated differently because 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, several attributes are defined along with the value. These are:

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

The function of each attribute is explained in the following sections.

4.3.2 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
- `.l` for long.

The suffix is not used when the label is referred to. Using of any suffixes other than `.b`, `.w` and `.l` results 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
```

4.3.3 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 4.4* on page 26.

4.3.4 Label scope

Often, in multi-module programs, a piece of code needs 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 are 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
```

4.3.5 Opcodes

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.

Opcodes must be separated from the preceding field (that is, label, if there is one) by a space or a tab. A comprehensive Opcode description can be found in the ST programming manual.

Macros are discussed in *Section 4.5* on page 30.

Directives are discussed in *Chapter 8: Librarian on page 51*.

4.3.6 Operands

Operands may be any of the following:

- Numbers and addresses.
- String and character constants.
- Program counter references.
- Expressions.

The following paragraphs 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 number 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 *Appendix A: Assembler directives* on page 55.

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 5* on page 23 shows a summary of these formats.

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

Table 5. 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**.
For example:
`"This is an ASCII string"`

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 from 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 6*) take precedence over operators in level #2 (listed in *Table 7*), and so on. In each level, operators have same precedence, they are evaluated from left to right.

Table 6. 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
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 7. Level 2 operators

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

Table 8. Level 3 operators

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

Table 9. 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 curly braces { } around arithmetic expressions.

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
```

4.3.7 Comments

Comments are preceded by a semicolon. Characters following a semicolon are ignored by the assembler.

4.3.8 A source code example

Below is an example of a short source code.

```
st7/
; small example module showing source formats
ioport      equ $8000      ; 8 bit I0 port A
handshake   equ $9000      ; write xx here to strobe

            segment 'program'
start       ld a,#0        ; zero counter
loop        ld ioport,x    ; store into ioport

            segment word at: FFFC 'code'
            WORD start
            end
```

Do not worry if some directives do not make sense yet; they will be covered soon. Also, take special notice of the SEGMENT directive.

4.4 Segmentation

4.4.1 Segments explained

Segments are very important. You have to understand segments before you can use the assembler. Take the time to understand them now and you will save yourself a lot of puzzling later.

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. Up to 128 different segments may be defined in each module. The segment directive itself has four arguments, separated by spaces:

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

For example:

FILE1:

```
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
```

FILE2:

```
st7/
            segment 'RAM0'
serialtemp  ds.b
serialcou   ds.b
            WORDS
            segment 'eprom'
serial_in   ld A,#0
            end
```

In the preceding example, FILE1 and FILE2 are two separate modules belonging to the same program. FILE1 introduces two classes: **'RAM0'** and **'eprom'**. The class-names may be any names you choose up to 30 characters.

The first time a class is used, introduced, 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 FILE2.

4.4.2 Parameters

Possible arguments are:

- *Name*
- *Align*
- *Combine*
- *cod parameter, output file control*

The following paragraphs describe each argument in detail, and the final paragraph describes *Copying code*.

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.

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 10* are allowed to be specified overriding the default.

Table 10. 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 26*, 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.

Combine

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

Table 11. 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.

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 cannot 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'
   ...
```

`cod` parameter, output file control

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 is 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
```

Copying code

It sometimes happens that you need to copy a block of code from EPROM to RAM. This presents some difficulties because all labels in that piece of code must have the RAM addresses, otherwise any absolute address references in the code will point back to the EPROM copy.

In this case, it helps to specify a class for **execution**, and use a different class for **storage**, as in the following example:

```
        segment byte at: 0 'code'
        segment byte at: 8000 'ram'
        segment 'ram>code'
label1:nop
```

The code starting from **`label1`** will be stored in the **`code`** class as usual, but all the labels in that special segment will be given addresses in the **`ram`** class, and memory will also be reserved in the **`ram`** class for the contents of the special segment.

4.5 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 is 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.

4.5.1 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 of 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
```

Note: ***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 12* are very useful, in conjunction with macros:

Table 12. 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.

4.5.2 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.

4.6 Conditional assembly #IF, #ELSE and #ENDIF directives

Conditional assembly is used to choose to ignore or select whole areas of assembler code. This is useful for generating different versions of a program by setting a particular variable in an **INCLUDE** file that forces the use of certain pieces of code instead of others.

There are three main directives used to perform conditional assembly, as shown in *Table 13*.

Table 13. Summary of conditional assembly directives

Directive	Usage
#IF	Marks the start of the conditional and decides whether the following zone will be assembled or not.
#ELSE	Optionally reserves the condition of the previous #IF for the following zone.
#ENDIF	Marks the end of the previous #IF's.

The condition given with the **'#IF'** may take the form of any numeric expression. The rule for deciding whether it resolves to 'true' or 'false' is simple: if it has a zero value then it is false, else it is true. These directives should NOT start at column 1 of the line, reserved for labels. For example:

```
#IF {count eq 1}
%OUT 'true'
#ELSE
%OUT 'false'
#ENDIF
```

This sequence would print **true** if the label **count** did equal 1, and '**false**' if it did not. For example:

```
#IF  {count gt 1}
%OUT count more than one
#IF  {count gt 2}
%OUT ...and more of TWO !
#ELSE
%OUT ...but not more than two!
#ENDIF
#ELSE
%OUT count not more than one
#ENDIF
```

As you can see, conditionals may be nested, the **#ELSE** and **#ENDIF** directive are assumed to apply to the most recent unterminated **#IF**.

Other special **#IF** directives are available as shown in *Table 14*.

Table 14. Other special #IF directives

Directive	Usage
#IF1 and #IF2	Requires no conditional argument. If the appropriate pass is. being assembled, the condition is considered '**true**'; for instance **#IF1** will be considered true while the assembler is in first pass, **#IF2** while in the second pass.
#IFDEF	Checks for label definition.

Table 14. Other special #IF directives

Directive	Usage
#IFB	Checks for empty argument (that is, empty, or containing spaces / tabs), useful for testing macro parameter existence.
#IFF	(IF False) is similar to #IF, but checks the negation of the condition argument.
#IFIDN	Tests for string equality between two arguments separated by a space. This is useful for testing macro parameters against fixed strings.
#IFLAB	Checks if the argument is a predefined label.

4.7 Running the assembler

4.7.1 Command line

The assembler needs the following arguments:

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

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

```
ASM
STMicroelectronics - Assembler - rel. 4.44
File to Assemble: game
```

In the example above, no parameters were given on the command line, so all the parameters were prompted for.

The **<file to assemble>** parameter assumes a default suffix **".ASM"**. For example, if you type '**game**' then '**game.asm**' is the actual filename used.

The listing file is the file to which the assembly report is sent if selected. The default filename (which is displayed in square brackets), is made from the path and base-name of the file to assemble. The default filename suffix for the assembly report file is "**.LST**". For instance, if you type '**game**', then '**game.lst**' is the actual filename used.

Note that unless the assembler is told to create either a pass-1 or pass-2 complete listing by the options argument, the listing file will not be created.

4.7.2 Options

Options are always preceded with a minus sign '-'. Upper and lower cases are accepted to define options. Supported options are listed in *Table 15*.

Table 15. Command line options

Option	Function
-SYM	Enable symbol table listing (see *page 34*)
-LI -LI=<listfile>	Enable pass-2 listing (see *page 34*) Enable listing and specify name of list file
-OBJ=<path>	Specify .OBJ file (see *page 34*)
-FI=<mapfile>	Specify 'final' listing mode (see *page 35*)

Table 15. Command line options

Option	Function
`-D <1> <2>`	#define <1> <2> (see page 36)
`-I`	Specify paths for included or loaded files (see page 36)
`-M`	Output make rule (see page 37)
`-PA`	Enable pass-1 listing (see page 37)
`-NP`	Disable phase errors (seepage 37)

SYM option

Description: 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. To specify the pathname for the generated list file use the option `-li=<pathname>`. The default extension is LST. Note that the extension must be three characters long.

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`

`ASM prog -li=prog.lsr`

The output is the file `prog.lsr`

OBJ option

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

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

Example: `ASM prog -obj=obj\prog`

Forces the assembler to generate the object file `obj\prog.obj`.

FI option

Note: ***Instead of using ASM -fi, it is advised to use the list file post processor ABSLIST which guarantees that the final list file is consistent with the executable code generated by the linker.***

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 set 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 linked your application to compute relocations and produce a `.COD` file and a map file.
To generate a full listing, 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.

Example:

```
ASM ex1                 (produces ex1.obj)
ASM ex2                 (produces ex2.obj)
LYN ex1+ex2,ex          (produces ex.map, ex.cod)
                        (see Chapter 5: Linker on page 38)
ASM ex1 -fi=ex.map      (produces ex1.lst)
ASM ex2 -fi=ex.map      (produces ex2.lst)
```

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

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: Allows 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, `EPROM` is replaced with `2`, `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.

I option

Description: Used to specify the list of search paths for files that are included (with #include) or loaded (with #load). The 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 ST 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: Tells the ST assembler to output a rule suitable for make, describing the dependencies to make an object file.
For a given source file, the ST 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
```

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`

NP option

Description: Disables the error generation.

Format: `ASM <file> -np`

Example: `ASM file1 -np`

5 Linker

5.1 What the linker does

After having separately assembled all the component modules in your program, the next step is to link them together into a **.COD** file which can then be sent on to its final destination using **OBSEND.**

This linking process is not just as a simple concatenation of the object modules. It resolves all the external references. If a referenced label is not defined as **PUBLIC**, an error is detected. It also checks the type of relocation to do, places the segment according to your mapping, and checks if any of them is overrun.

5.2 Invoking the linker

5.3 Command line

5.3.1 Arguments

The linker needs the following arguments:

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

-no_overlap_error forces the generation of the **.cod** executable even if some segments overlap.

If all or any arguments are left out of the command line, you will be prompted. For example:

```
LYN
STMicroelectronics - Linker - rel  3.00
.OBJ files: begin
.COD file [begin.cod]: begin
Libraries:
```

The **.OBJ** files are simply 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.

The **.COD** file has a default name formed of the first object file's name with forced suffix of **.COD**. This will be the name of the file produced at the end of the link session. It contains all the information from the link session in a special format: however, **OBSEND** must be used on the **.COD** file before it is ready to use. If the default filename is not what you want, the filename given at the prompt is used instead. The suffix will be forced to **.COD** if left blank. The default is selected by leaving this argument blank at the command line, or pressing **<ENTER>** at the prompt.

The **Libraries** prompt asks for a list of library files generated by the lib utility that should be searched in case of finding unresolved external references. The format for giving multiple libraries is the same as for the **.OBJ** list, except the suffix **.LIB** is assumed.

Some examples:

Linking together the modules **game.obj**, **scores.obj**, **key.obj**, **game1.obj, game2.obj** and **game3.obj** without using any libraries and generating a **.COD** file named **game.cod**, requires the following command line:

```
LYN game+scores+keys+game1+game2+game3;
```

Linking the same modules in the same environment, but generating a **.cod** file named **prog.cod** requires the following command line:

```
LYN game+scores+keys+game1+game2+game3,prog;
```

5.3.2 Response files

Response files are text files that replace the command line to generate the arguments required. Although they can be used on the assembler and linker, it only really makes sense to use them on the linker.

The command line given with the name of the program to execute (here **LYN**) can only take up to 128 characters as its argument. For most programs this is fine, but the linker allows up to128 modules to be linked in one run; all their names have to be declared to the linker in its first argument.

This is where response files come in, **they allow you to redirect the command line parser to a file** instead of expecting arguments to come from the command line or the keyboard. A response file is invoked by giving an '**@**' sign and a filename in response to the first argument you want to come from the response file.

The filename is assumed to have a suffix '**.RSP**' if none is supplied. Repeating our example used as earlier, but this time with a response file called **game.rsp**:

```
LYN @game.rsp
```

is all that needs to be typed, and the file **game.rsp** must contain:

```
game+scores+keys+

game1+

game2+game3

prog
```

Which echoes what would have been typed at the keyboard. If the response file ends prematurely, the remaining arguments are prompted for at the keyboard. In very large session, the **.OBJ** files argument will not fit on one line: it can be continued to the next by ending the last **.OBJ** file on the first line with a '**+**'.

Note: When using response files, there must be at least two carriage returns at the end of the file.

5.4 Linking in detail

5.4.1 PUBLICs and EXTERNs

All labels declared external in the modules being linked together must have a corresponding **PUBLIC** definition in another module. If it does not, it may be an error. Similarly, there must only be one **PUBLIC** definition of a given label.

The bulk of the linker's job is filling those relative or external blanks left by the assembler in the **.OBJ** files; to a lesser extent, it also handles special functions such as **DATE** or **SKIP** directives. Equally important, it has to collate together and allocate addresses to segments.

5.4.2 Segments in the linker

A typical system may look like the diagram alongside: a good candidate for four different segments, perhaps named RAM0, RAM1, EPROM and ROM.

If the reset and interrupt vectors live at the end of the map, perhaps from FFEE-FFFF then we might mark a fifth segment called vectors at those addresses and truncate ROM to end at FFED; that way the linker will warn us if ROM has so much code in it that it overflows into where the vectors live. These classes would be introduced as follows:

```
segment byte at: 0-FF        'RAM0'
segment byte at: 100-027F    'RAM1'
segment byte at: 8000-BFFF   'EPROM'
segment byte at: C000-FFDF   'ROM'
segment byte at: FFE0-FFFF   'VECTORS'
```

After their full introduction that needs only be done once in the whole program, the rest of the program can refer to the classes just by giving the class names in quotes, for example:

segment 'RAM0'

```
xtemp   ds.w      ; temp storage for X register
time    ds.b      ; timer count index
```

segment 'ROM'

```
hex     ld A,#1
        add A,#10
        nop
```

If this example followed immediately after the class instruction the **'xtemp'** label would be given the value **0**, **time** would be given **2** and **hex C000**. If, however, the code was several modules away from the introduction with segments of the classes '**RAM0**' or '**ROM**', then the value allocated to all the labels will depend on how much space was used up by those modules. The linker takes care of all this allocation. This is the way the linker handles the problems of relocatability; keep in mind that this link system is going to have to handle compiled code from high level languages and you will perhaps begin to understand why things have to be generalized so much.

So far the segments we have looked at have had no **<name>** field, or, more accurately, they all had a null **<name>** field. You can ensure that related segments of the same class, perhaps scattered all over your modules with segments of the same class are collated together in a contiguous area of the class memory by giving them the same name.

For example:

```
grafix      segment byte at: 100-027F 'RAM1'
cursor_buf  ds.b 64          ; buffer for map under cursor
            segment byte at: 8000-BFFF 'ROM'
show_page   nop
            segment 'RAM1'
field-buf   ds.b {{256 mult 256}/8}
            segment 'ROM'
dump_buf    ld A,field_buffer
grafix      segment 'RAM1'
cursor_temp ds.b 64
```

This complex sequence of segments shows now instances of the class **RAM1** being used with a segment name of **grafix**. Because the first instance of the class **RAM1** had the name **grafix** the two **grafix** RAM1 segments are placed in memory first followed by the null-name RAM1 segment (which defines **field_buf**). Note this is not the order of the segments in the code, segments with the same name are collated together (even from separate **.OBJ** files), and the lumps of segments of the same name are put into memory in the order that the names are found in the **.OBJ** files.

As explained on *page 29*, if **x** is your cod file suffix when introducing a class, all code for that code is sent into a new cod-file named **file_x.cod**, where **file** is the name of the first cod file, and **x** is the cod-file suffix (**1-9**).

5.4.3 Symbol 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.

The linker supports a special feature, you can link in **.SYM** files from other link sessions. This means that with big programs, you cannot only partition your code at assembler level, but divide the code up into 'lumps' which are linked and loaded separately, but have access to each other's label as **EXTERNs**. You can 'link in' a symbol table simply by giving its name with the suffix **.SYM**. Always give symbol tables at the start of the object file list.

OBJ file example: `LYN prog1.sym+prog2,vectors,irq;`

Once this is done, all the **PUBLIC** symbols from **prog1.sym** are now available as **PUBLICs** to **prog2.obj**, **vectors.obj** and **irq.obj**.

Because changes in one link will not automatically update references to the changed link code in other links, it is necessary when using this technique to 'fix' each link in an area of memory, and have a 'jump table' at the top of each area. This means that all 'function' addresses are permanently fixed as jump table offset, and changes to each link will result in automatic redirection of the jump targets to the new start of each routine. Put another way, each link must have entry fixed points to all its routine, otherwise re-linking one 'lump' of a program could make references to its addresses in other modules out of date.

5.5 The linker in more detail

5.5.1 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 assembler has been so liberal with. 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.

5.5.2 The composition of the .COD files

The **.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. It only remains for **OBSEND** to take the code segment by segment and send it on to its destination.

5.5.3 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, that is, 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 this 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
```

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.

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

6 OBSEND

6.1 What OBSEND does for you

After your program has been assembled and linked to form a **.COD** file it must be sent to the place where it will be executed. Right now, your code is just stored as a file on a disk where the target system cannot get at it.

OBSEND is a general purpose utility for **.COD** files in various ways using various formats.

6.2 Invoking OBSEND

OBSEND follows the same standard formats as the rest of the assembler / linker; arguments can be given from the command line, keyboard or response file. The general syntax is:

```
OBSEND <file>,<destination>[,<args>],<format>
```

where **<file>** is the name of the **.COD** file to be formatted (default extension **.COD**). If the filename is not given on the command line, you are prompted at the keyboard with:

```
OBSEND
STMicroelectronics - Obsend - rel. .2.00
File to Send: test
Destination Type (<f>ile,<v>ideo): f
Final Object code Filename [test.fin]: test.s19
Object Format <ENTER>=Straight Binary, ...,
       ST REC <2>, ST REC <4>: s
```

6.2.1 Destination type

<destination> can be **f** (file) or **v** (video). Only a single character is required.

6.2.2 Destination arguments

When the destination type is **f** (file) the argument **<filename>** tells OBSEND where to send the code. The default suffix **.FIN** is assumed if none is given. For example:

```
OBSEND test,f,image.s19,s
```

The command generates the file **image.s19** containing the code from **test.cod**, in S-record **s** format.

When the destination code is "**v**" (video), this field is void.

6.2.3 Format definitions

<format> specifies the output format. Output format options are listed in *Table 16*.

Table 16. Output formats

<format>	Output format
<none>	straight binary, that is, a bit-for-bit image
i	Intel hex

Table 16. Output formats

<format>	Output format
i32	Intel hex with 32 bytes of data per line
ix	Intel hex extended
s	Motorola S-record (1 byte per address, for example ST7)
x	Motorola S-record extended with symbol file
2	ST S-record 2 (2 bytes per address, for example D950)
4	ST S-record 4 (4 bytes per address, for example ST18932 program space)
f	'Filled' straight binary format
g	GP industrial binary format

6.2.4 Straight binary format

`<format>= <none>`

This is the simplest of the formats. It is nothing but a bit-for-bit copy of the original file. This is the usual mode for sending to the EPROM emulators, etc., and is the default if no format argument is given.

*Note: When the destination is the screen (the destination code is "**v**"), do not use this format; otherwise you get weird control codes.*

`<format>= <f>`

This is the 'filled' straight binary format where gaps between adjacent segments are filled with **$FF**.

6.2.5 Intel hex format

`<format>= i`

This format is very much more complex. Intel hex bears similarities to S-record that we look at later. Let's look at a line of the Intel hex format in detail:

```
:10190000FFFFFFFFFFC00064FFC0006462856285E0
10   number of data bytes (16 in decimal)
1900 address
00   record type
...  data bytes
E0   checksum
```

The first thing to note is that everything is in printable ASCII. Eight-bit numbers are converted into two-character hexadecimal representation.

Each line begins with an ASCII '**:**' (**$3A**) character.

The next two characters form a byte that declares how many data bytes follow in the data byte section a little further along.

The next four characters form a 16-bit high-byte first number that specifies the address for the first byte of this data; the rest follows on sequentially.

The next two characters are the record type for this line: **00** is a data line, and **01** signals EOF. The following characters, until the last two, are the 16 data bytes for this line, the last two are a checksum for the line, calculated by starting with **$00** subtracting the real value of all characters sent after the '**:**' until the checksum itself. 'Real value' means that for example, the two characters **3** and **0** should subtract **$30** from the checksum, not **51** and **48**. Every line ends with a CR-LF combination, **$0A** and **$0D**.

The last line sent must be an END-OF-FILE line, which is denoted by a line with no data bytes and a record type of 01 instead of 00.

Giving I32 or i32 instead of intel as the argument uses the same format, but sends 32 bytes of data per line.

6.2.6 Motorola S-record format

```
<format>= s
```

This is another complex method for sending data. Again it cuts the data into 16-byte 'records' with overhead both sides. S-record come in four types: **S0**, known as a header record, **S1** and **S2** data records with 16 and 24-bit address fields, and **S9** and **S8** EOF records with 16 and 24-bit address fields.

*Note: The convention is to close an **S1** 16-bit data record with the **S9** 16-bit EOF record, and to close an **S2** 24-bit data record with the **S8** 24-bit EOF record.*

```
S10D0010E0006285E000628562856D
S1   record type
0D   number of bytes left,address,data and checksum (13 in decimal)
0010 address
.... data bytes
6D   checksum
```

The first two characters define the record type: **S0**, **S1**, **S2**, **S8** or **S9**.

The next two characters form a hexadecimal representation of the numbers of bytes left in the record (that is, numbers of characters /2) This count must include the checksum and addresses bytes that follow. The address field is four characters wide in **S0**, **S1**, **S9** and six characters wide in **S2** and **S8**. The most significant character always comes first.

OBSEND always uses **S1** type records wherever possible (that is, when the address is less than **$10000**) and use **S2** type data records where it has to (that is, address > **$FFFF**).

Up to 16 data bytes then follow, with the checksum appended on the end. The checksum is calculated by starting with **$FF** and subtracting the 'real value' of all bytes sent from and including the byte count field until the checksum itself. In this context, 'real value' means the value of the byte before it is expanded into two ASCII characters.

The record is concluded by a CR-LF combination **$0A**, **$0D**. The **S0**, **S8** and **S9** (that is, header and EOF) records are always the same:

```
S00600004844521B
```

and:

```
S804000000FB
S9030000FC
```

A complete example of S-record transmission may look like:

```
S00600004844521B
```

```
   S113001AFF120094FF130094D08AFF390094FF1250
   S20801C004FFC0000073
<format>= x
```

The **extended S-record format**, selected by format x, sends code as described above, except that after the **S9**, it sends a list of **SX** records, one after the other, in the format:

```
   SX  0000 LABEL
```

where **0000** are four ASCII zeroes, and LABEL is five ASCII characters. There are two spaces after the **SX** and one space after the **0000**. **0000** represents the hexadecimal value of the label. LABEL may extend to 31 characters, and end with a carriage return.

6.2.7 ST 2 and ST 4 S-record formats

```
<format>= 2
```

```
<format>= 4
```

These are industrial formats defined for specific needs:

- 2: specify 2-byte words for one address.
- 4: specify 4-byte words for one address.

6.2.8 GP binary

```
<format>= g
```

This format is simple. It has a 16-byte count at the beginning low-byte first, calculated by starting at **0**, and adding the value of each byte until the end of the data is reached. If there are any 'gaps' in your code, OBSEND fills them in with **$FF**, and adjusts the checksum accordingly. After four bytes of header information, the data follows in one big block.

7 ABSLIST

7.1 Overview

As the list file with absolute addresses generated by the assembler from the source file and the map file (ASM ex1.asm -fi=ex.map) may show differences with the actually generated code, a post processor has been written to be sure that the list file will be coherent with the executable file.

ABSLIST is a post processor which reads a list file with relative addresses and unresolved symbols and converts it into a list file with absolute addresses and resolved symbols. For this, the post processor needs information which is located in two files generated by the linker: the map file and the executable file in Motorola S-record format (.s19) or in Intel Hex format (.hex).

This is possible because the linker does not optimize the code generated by the assembler.

The list file with relative addresses is generated by the assembler and it must include symbols.

Thus the following assembler command must be executed first, to generate a list file with relative addresses and including a symbol list:

```
asm -sym file1.asm -li=Debug\file1.lsr
```

Such a list file is composed of two parts:

- A list of assembler instructions with addresses, codes and mnemonics,
- A list of labels.

To transform relative addresses for instructions and labels, the postprocessor adds to the relative address the start address of the corresponding segment.

The segment start address is found in the segment list of the map file.

As for the list of relative labels, there are two cases:

- Public labels: their absolute addresses can be found in the external label list of the map file.
- Private labels: as for the instructions, the start address of the corresponding segment must be added to the relative address.

The segment corresponding to an instruction is the last segment which has been declared in the source file.

It is the same for a local label, so a list of labels with the segments where they are defined must be constituted as the list file is parsed.

To generate the code for instructions with unresolved labels (subroutine calls, variable read or write accesses), the final code is read in the executable file.

7.2 Invoking the list file post processor

Here is the full command syntax of the list file post processor:

```
abslist <rel_list_file> -o <abs_list_file> -exe
<application>.(s19|hex) -map <application>.map

<rel_list_file> ::= <file>.lsr
<abs_list_file> ::= <file>.lst
```

-o precedes the output list file.

-exe precedes the executable file name. The executable format can be Motorola S-Record format or Intel Hex format. The format is recognized by reading the first line of the executable file.

-map precedes the map file name.

`-o` and `-map` options may be omitted.

If `-o` is omitted, the absolute list file name is deduced from the relative list file name by replacing its extension with .lst.

If `-map` is omitted, the map file name is deduced from the executable file name by replacing its extension with .map.

Here is the reduced command syntax:

```
abslist <rel_list_file> -exe <application>.(s19|hex)
```

It is possible to convert several list files at the same time. The source file names must be separated by "," with no blank in between. If several source file names are given and if `-o` option is used, corresponding destination file names must also be given.

For example:

```
abslist <rel_lst_file1>,<rel_lst_file2>,...,<rel_lst_fileN>
-o <abs_lst_file1>,<abs_lst_file2>,...,<abs_lst_fileN>
-exe <application>.s19
```

Example:

```
ASM -sym -li=ex1.lsr ex1.asm                (produces ex1.obj and ex1.lsr)
ASM -sym -li=ex2.lsr ex2.asm                (produces ex2.obj and ex2.lsr)
LYN "ex1.obj+ex2.obj,ex.cod;"               (produces ex.cod and ex.map)
OBSEND ex.cod,f,ex.s19,s                    (produces ex.s19)
ABSLIST ex1.lsr -o ex1.lst -exe ex.s19      (produces ex1.lst)
ABSLIST ex2.lsr -o ex2.lst -exe ex.s19      (produces ex2.lst)
```

Or

```
ABSLIST ex1.lsr,ex2.lsr -exe ex.s19         (produces ex1.lst and ex2.lst)
```

7.3 Limitations

1. There is one main limitation. The update of the relative address is based on the search of the last declared segment. If the search cannot succeed because of the use of `.NOLIST` directives which hide segment declarations, the absolute file cannot be properly generated.
   There is the same problem with `.XALL` and `.SALL` for macro expansions. If a segment is declared in a macro, these directives should not be used.
   There is the same kind of problem for label definitions which are removed from the list file by the previously mentioned directives. Label definitions are needed to compute the addresses of labels printed in the symbol table at the end of the list file.
   In conclusion, do not use `.NOLIST`, `.XALL` and `.SALL` primitives to hide code where segments are declared or labels are defined.
2. There is another limitation regarding the use of the `EQU` and `CEQU` directives to define private labels.
   The addresses of public labels can be found in the symbol table generated in the map file.
   There is a problem with private labels set by `EQU` or `CEQU` to a relative expression involving a label defined in a relative segment. As ABSLIST does not parse the expression after `EQU` and `CEQU` directives, it has no way to know which label is used in the expression and which segment it belongs to.
   ABSLIST always generates warnings for labels equaled to relative expressions.
   For private labels equaled to relative expressions, the post processor will print question marks for the unknown address.
   There is a workaround to get the addresses of labels equaled to relative expressions in the list file: just make these labels public and ABSLIST will be able to find their addresses in the map file.
3. ABSLIST only accepts the ST7 and the STM8 processors. It could be easily generalized to other processors but more validation time would be necessary.

8 Librarian

8.1 Overview

If you do a lot of work on similar boards especially those with the same processor, it makes a great deal of sense to reuse lumps of code you have already written to do the same task in a different program. At the simplest level, you could just copy the source code as a block of text into the new program. This works fine, but has a subtle disadvantage: if you update the subroutine, you have to hunt around all the usages of it, performing the update on each.

To get around this problem, many people have the source for common routines in one place, and link the **.OBJ** module with each program needing routine. Then you only need to update the source code once, reassemble it to get a new **.OBJ** file, then link again all the users of the routine, who will now have the new **.OBJ** file.

While this scheme works well, it generates some problems of its own. For example, each routine needs its own **.OBJ** file. By nature, these common routines tend to be small, so you end up giving dozens of extra **.OBJ** modules to the linker, and having the **.OBJ** modules scattered around your hard disk.

The base concept of a librarian is to combine all these small, useful **.OBJ** modules into one large **.LIB** library file. You could then tell the linker about the library, and it takes care of which **.OBJ** modules to pull in to link. It would know which ones to pull in by the fact that the main code being linked would have undefined externals, for example, to call the missing library routines. The librarian simply takes each undefined external in turn, and checks it against all the modules in the library. If any of the modules declares a **PUBLIC** of the same name, it knows you need that **.OBJ** module and it includes it automatically.

8.2 Invoking the librarian

The librarian is called **LIB**, and takes one command line argument that is the name of the library to operate on. If not given, you are prompted for it.

```
LIB [library name]
```

.LIB is added if the suffix is left off.

If the library you indicate does not exist, LIB asks you if it is a new library. For example:

```
LIB LIB1
STMicroelectronics - Librarian - rel 1.00
Couldn't open Library file 'LIB1.LIB'
is it a new file? (y/n): y
```

If the answer is '**n**', LIB aborts. If the library exists, LIB prints up a report on the library.

```
Library LIB1.LIB is 2K long.
16/1024 Public labels used in 2/128 modules.
```

Next comes the main prompt:

```
LIB1.LIB: Operation (<ENTER> for help):
```

Pressing **ENTER** gives you access to the options shown in *Table 17*.

Table 17. Library file options

Operation	Description
`+filename`	Add/update object module to/in library
`-filename`	Delete object module from library
`!filename`	Update object module in library
`*filename`	Copy object module to separate file from library
`?`	List contents of library
`x`	Exit to DOS

8.3 Adding modules to a library

Typing for example: `+user1\board` would look for a file, called **`user1\board.obj`**, and add it to the library.

If LIB cannot find the named file, LIB reports the fact and returns to the operations prompt. Else LIB issues the following message:

```
Adding new board.obj ...
15 labels added
Done.
```

If the library already contains a file **`board.obj`**, it prompts you with:

```
board.obj already in library LIB1.LIB,
replace with board.obj (Y/N):
```

Responding with '**`N`**' returns you to the operations prompt, while '**`Y`**' first removes the old **`board.obj`** then continues as above.

8.4 Deleting modules from a library

This is done by, for example:

```
-board
```

If LIB cannot find **`board.obj`** in the current library, it reports an error and aborts back to the operation prompt.

If it can find it, it makes sure you know what you are doing with:

```
board.obj to be deleted from library LIB1.LIB:Are you sure (Y/N):
```

`N`' aborts to operation prompt. '**`Y`**' continues, reporting:

```
Removing old board.obj ...
Done.
```

8.5 Copying modules from a library

To make a copy of a .OBJ module located in a library back to your hard disk, use, for example:

```
*board
```

This checks the existence of **board.obj** in the current library, if not it reports the failure and aborts the operation prompt. If it does find it, it invites you to give it the name of the hard disk file to create to contain the copy of the **.OBJ** module.

```
Copy into .obj file [board.obj]:
```

If you type <**ENTER**>, it selects the original name of the object module as the copy's name. Otherwise, give it a path spec. If the file you give already exists, LIB says:

```
File board.obj already exists; overwrite? (Y/N):
```

Again, responding '**N**' aborts to the operations prompt, while '**Y**' does the copy with the message:

```
Copying board.obj to disk...
Done.
```

8.6 Getting details in your library

The last operation:

```
?
```

causes LIB to print out details of the current library.

```
Library LIB1.LIB is 2K long
16/1024 Publics labels used on 2/128 modules
0: z1.obj (z1.asm) length 2DE
1: board.obj (board.asm) length 7FFF
```

The name in brackets is the source module from which the named object module was assembled.

9 Definitions

Table 18. Acronyms and terms used in this document

Name	Definition
Application board	This is the printed circuit board onto which you wish to connect the target ST MCU. It should include a socket or footprint so that you can connect the application board to your emulator or development kit using the probe and the appropriate device adapter. This allows you to emulate the behavior of the ST MCU in a real application in order to debug your application program.
Device adapter	Device adapters are included in your emulator kit to allow you to connect the emulator to your application board. The type of device adapter depends on the target device's packaging. Many MCUs come in at least different packages, and you should therefore use the device adapter that corresponds to the type of package you have chosen for your application.
DIL	Dual in line. Designates a type of device package with two rows of pins for thru-hole mounting. Sometimes also called DIP (dual in-line package).
ECP	Extended capabilities port communication standard.
EPP	Enhanced parallel port communication standard.
LSB	Least significant byte of a 16-bit value.
Main board	This is the main board of the emulator that is common to the entire ST HDS2 family of emulators. It controls common functions such as communication with your PC via the parallel port.
mem	Memory location.
mnem	Mnemonic.
MCU	Microcontroller unit. Otherwise referred to as the target device throughout this manual. This is the core product (or family of products) for which the Development Kit is designed to act as an emulator and programming tool. In general terms, an MCU is a complete computer system, including a CPU, memory, a clock oscillator and I/O on a single integrated circuit.
ST7MDT6-active probe	A printed card having connector pins that allow you to connect the emulator to the MCU socket of the user application board. Using the active probe allows the HDS2 emulator to function as if it were the target device embedded in your application. The probe is connected to the emulator by two flat cables.
PC	The program counter is the CPU register that holds the address of the next instruction or operand that the CPU will use.
S	Stack pointer LSB.
short	Uses a short 8-bit addressing mode.
SO	Small outline. Designates a type of device package with two rows of pins for SMD or socket mounting. For example, SO34 designates a 34-pin device of this package type.
src	source
ST7 visual debug (STVD7)	A graphic debugger software package that allows you to debug applications destined for the ST7 family of MCUs, either using a built-in simulator function, a Development Kit or an HDS2 Emulator.
Target device	This is the ST MCU that you wish to use in your application, and which your emulator or development kit will emulate for you.
User application board	Designates your application board.

Appendix A Assembler directives

A.1 Introduction

Each directive is described in a table.

- The name of the directive is given in the table title (and always appears in the Format).
- The Format shows the arguments allowed (if any) for this directive.
- The Description describes the action of the directive and the format and nature of the argument specified in the Format.
- The Example gives one or more example of the directive in use.
- The See also lists possible cross references.

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

Table 19. List of directives

Directive	Table	Directive	Table	Directive	Table
.BELL	*Table 20*	GROUP	*Table 41*	.NOCHANGE	*Table 62*
BYTE	*Table 21*	#IF	*Table 42*	.NOLIST	*Table 63*
BYTES	*Table 22*	#IF1 Conditional	*Table 43*	%OUT	*Table 64*
CEQU	*Table 23*	#IF2	*Table 44*	.PAGE	*Table 65*
.CTRL	*Table 24*	#IFB	*Table 45*	PUBLIC	*Table 66*
DATE	*Table 25*	#IFIDN	*Table 46*	REPEAT	*Table 67*
DC.B	*Table 26*	#IFDEF	*Table 47*	.SALL	*Table 68*
DC.W	*Table 27*	#IFLAB	*Table 48*	SEGMENT	*Table 69*
DC.L	*Table 28*	#INCLUDE	*Table 49*	.SETDP	*Table 70*
#DEFINE	*Table 29*	INTEL	*Table 50*	SKIP	*Table 71*
DS.B	*Table 30*	INTERRUPT	*Table 51*	STRING	*Table 72*
DS.W	*Table 31*	.LALL	*Table 52*	SUBTTL	*Table 73*
DS.L	*Table 32*	.LIST	*Table 53*	.TAB	*Table 74*
END	*Table 33*	#LOAD	*Table 54*	TEXAS	*Table 75*
EQU	*Table 34*	LOCAL	*Table 55*	TITLE	*Table 76*
EXTERN	*Table 35*	LONG	*Table 56*	UNTIL	*Table 77*
#ELSE	*Table 36*	LONGS	*Table 57*	WORD	*Table 78*
#ENDIF	*Table 37*	MACRO	*Table 58*	WORDS	*Table 79*
FAR	*Table 38*	MEND	*Table 59*	.XALL	*Table 80*
FCS	*Table 39*	MOTOROLA	*Table 60*	ZILOG	*Table 81*
.FORM	*Table 40*	NEAR	*Table 61*		

A.2 Directives

Table 20. .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	

Table 21. 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 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: theses are translated into ASCII and processed byte by byte. It is generally used for defining data tables. 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

Table 22. 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 of the label), relativity (absolute or relative), and length, (BYTE, WORD and LONG). All of these attributes, except length, are defined explicitly before or at the definition. You can force the label to be a certain length by giving a dot suffix, e.g. 'label.b' forces it to be byte length. 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	`BYTES` `lab1 EQU 5 ; byte length for lab1`
See also	LONGS, WORDS

Table 23. 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

Table 24. .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

Table 25. 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 the assembler leaves spare at the position of the DATE directive. This means that every link will leave its date in the object code, allowing automatic version control. The date takes the form (in ASCII) DD_MMM_YYYY where character '_' represents a space; for example 18 JUL. 1988. 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	

Table 26. 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**.

Table 26. DC.B

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	

Table 27. 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

Table 28. 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

Table 29. #DEFINE

Purpose	Define manifest constant.
Format	`#DEFINE <CONSTANT ID> <real characters>`

Table 29. #DEFINE

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	

Table 30. 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; the 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

Table 31. DS.W

Purpose	Define word space in object code.
Format	`DS.W [optional number of words]`

Table 31. DS.W

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; the 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

Table 32. 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

Table 33. 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	

Table 34. 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	

Table 35. 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 a way of telling the assembler not to expect the 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, (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

Table 36. #ELSE

Purpose	Conditional ELSE.
Format	**#ELSE**
Description	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

Table 37. #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

Table 38. FAR (STM8 only)

Purpose	Specifies to debuggers that the return address in the stack for functions using this directive is written over three bytes.
Format	**FAR <"string">**
Description	This directive is used with functions called by CALLF, whose return stack address spans three bytes. Every function called by CALLF must be classified as FAR. This directive is for use with the STM7 Assembler only.

Example	`PUBLIC func`
	`FAR func`
	`func retf`

See also	NEAR, INTERRUPT

Table 39. FCS

Purpose	Form constant string.
Format	`FCS <"string"> |<bytes> [<"string"> |<bytes>]...`
Description	This directive works in the same way as the common STRING directive, except that the last character in any string argument has bit 7 (for example 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

Table 40. .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

Table 41. 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`
See also	

Table 42. #IF

Purpose	Start conditional assembly.
Format	`#IF <exp>`

Table 42. #IF

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

Table 43. #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

Table 44. #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

Table 45. #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

Table 46. #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

Table 47. #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

Table 48. #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

Table 49. #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 important differences: the first line usually reserved for the processor name is like any other source line, and they have no **END** directive. They are used to 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 goes off to the named INCLUDE file and assembles this file before returning to the line after the #INCLUDE directive in the former source code file. The benefit is that any alterations made to a macro must be done once, in the include file; but you'll still have to reassemble all modules referring to the changed entry. NOTE that the filename must be inside double-quotes.

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

See also	

Table 50. INTEL

Purpose	Force Intel-style radix specifier.
Format	`INTEL`
Description	The Intel style: `0ABh` Hexadecimal `17o or 17q` Octal `100b` Binary `17` Decimal (default) `$` Current program counter This directive forces the Intel format to be required during the assembly.

Table 50. INTEL

Example	`INTEL` `ld X,0FFFFh`
See also	MOTOROLA, TEXAS, ZILOG

Table 51. INTERRUPT

Purpose	Specifies to 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, FAR

Table 52. .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

Table 53. .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

Table 54. #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, that is, a direct image of the bytes you want in the object code. It should not be in Motorola (.s19) or Intel (.hex) format.

Table 54. #LOAD

Example	`segment byte at 8000-C000 'EPROM1'` `#LOAD "table.bin"`
See also	

Table 55. 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

Table 56. 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 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. **LONG** sends long words with the least significant byte first. It's generally used for defining 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

Table 57. LONGS

Purpose	Default new label length long.
Format	`LONGS`
Description	When a label is defined, four SEPARATE attributes are defined with it: scope (internally or externally defined), value (actual numerical value of the label), relativity (**absolute** or **relative**), and lastly, length (BYTE, WORD or LONG). All these attributes except length are defined explicitly before or at the end of the definition: you can force a label to be a certain length by giving a dot suffix, for example 'label.b' forces it to be byte length. 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

Table 58. 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. **Note**: If you don't want the definition of the macro to be listed, insert directive .NOLIST before the macro definition, and append directive .LIST after the 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	MEND, .LALL, .SALL, .XAL

Table 59. MEND

Purpose	End of macro definition.
Format	`MEND`
Description	End of macro definition.

Table 59. MEND

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

Table 60. 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

Table 61. 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	FAR, `INTERRUPT`

Table 62. .NOCHANGE

Purpose	List original #define strings.
Format	`.NOCHANGE`

Table 62. .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

Table 63. .NOLIST

Purpose	Turn off listing.
Format	`.NOLIST`
Description	Certain parts of your modules may not be required on a listing; this directive disables the listing until the next **.LIST** directive. The default is for the listing to be enabled. This directive, in conjunction with the directive **.LIST**, can be used to control the listing of macro definitions. This directive does not generate assembly code or data.
Example	`.NOLIST`
See also	LIST

Table 64. %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	

Table 65. .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	

Table 66. PUBLIC

Purpose	Make labels public.
Format	`PUBLIC <arg>`

Table 66. PUBLIC

Description	This directive marks out given labels defined during an assembly as **'PUBLIC'**, 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 won't need to 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

Table 67. 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

Table 68. .SALL

Purpose	Suppress all body of called macro.
Format	`.SALL`
Description	This directive forces the complete suppression of the listing of a macro expansion each time a macro is invoked. This instruction is never listed. **Note:**This directive may produce confusing listings.
Example	`.SALL`
See also	.LALL, .XALL

Table 69. 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:

Table 69. SEGMENT

Example	**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
See also	**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.

Table 70. .SETDP

Purpose	Set base address for direct page.
Format	**.SETDP <base address>**
Description	If you have used an ST processor, you are 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	

Table 71. 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	

Table 72. 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

Table 73. 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

Table 74. .TAB

Purpose	Set listing field lengths.
Format	`.TAB <label>,<Opcode>,<operand>,<comment>`
Description	Sets the size of the four source code fields for listings. The defaults of 0, 8, 16, 24 are for 80-column printer; if yours can go wider, you need to tell the assembler using this directive. The four fields are the width of the label field, the opcode field, operand and comment. This directive does not generate assembly code or data.
Example	`.tab 10,6,16,40`
See also	.LIST, .NOLIST

Table 75. TEXAS

Purpose	Texas Instruments-style radix specifier.
Format	`TEXAS`

Table 75. 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

Table 76. 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

Table 77. 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

Table 78. WORD

Purpose	Define word in object code.
Format	`WORD <exp>[, <exp>...]`

Table 78. WORD

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

Table 79. 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 these attributes except length are defined explicitly before or at the definition: you can force the label to be of a certain length by giving a dot suffix, for example '**label.b**' forces it to 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

Table 80. .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

Table 81. ZILOG

Purpose	Force Zilog-style radix specifiers.
Format	`ZILOG`

Table 81. 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 error trapped by the assembler

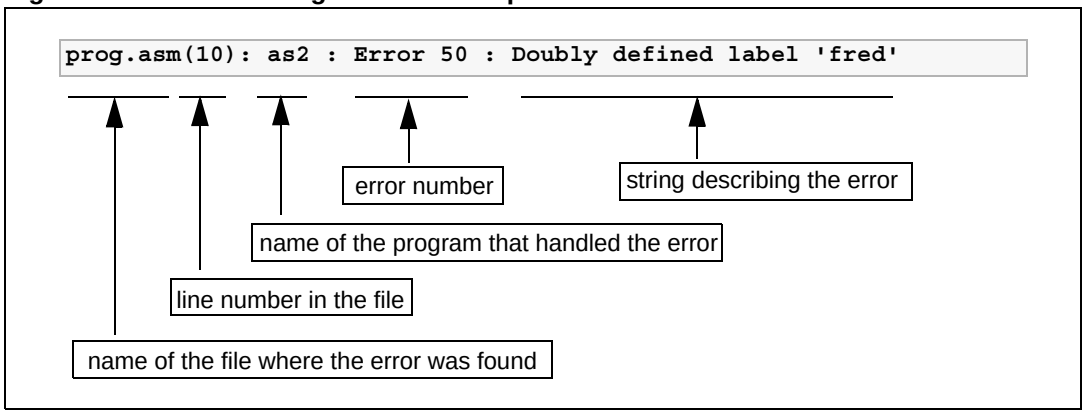
- fatal
- recoverable

A fatal error stops the assembly there and then, returning you to the caller (which may or may not be DOS; CBE can also invoke the assembler) with a message and error number describing the problem.

The format of the error messages is as follows:

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

Figure 3. Error message format example



*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

Both fatal and recoverable errors are copied into the file `CBE.ERR` as they occur. Most link errors (described on *page 83*) are also copied into `CBE.ERR`.

CBE can use this error file to give automatic error finding.

B.3 Assembler errors

Table 82. Assembler errors

Error	Definition
1	Empty file: The assembler could not read even the first line of the given source code file.
2	EOF while in Macro Definition: The file ended while a macro was being defined; you should end the last macro definition properly with a MEND statement.
3	Could not return to old place in source file 'X.asm': This error should never occur; it implies you have a disk fault of some kind. After a #include, the assembler returns to the line after the #include itself. If it cannot return to that line this error is produced.
4	Illegal source EOF: Main source code files must end with an END directive and a carriage-return.
5	EOF before line terminator: The END directive must have at least one CR after it: for example <TAB> END <EOF> will generate this error while <TAB> END <EOF> will work fine.

6	Code produced outside segment: Any code produced by the assembler is going to have to be placed on a given address in the target system at the end of the day. Since segments are the assemblers way of allocating addresses to lumps of code, any code generated before the first SEGMENT directive is nonsense.
	`^T 55 Move to top line of current window`
	`^V 56 Move to last line of current window`
	`^U 57 Undo changes to last edited line`
	`^[ 58 Drop start of black marker`
	`^] 59 Drop end of black marker`
	`^F 60 Find Source Code for Hex address`
	`^J 61 Report value of given label`
	`^N 62 Report address of current Editor line`
	These functions mostly explain themselves; **the Alternate functions do the same job as the original functions of the same name:** having two indexes for the same job allows the cursor keys and control codes to move the cursor, whichever the user prefers. Some indexes are not used by the default key sequence matrix; these allow some WordStar-like commands to be implemented with more meaning.
	Multiple-key sequences, such as those found in WordStar format control codes need to be implemented as follows: take the sequence <^Y><L>, that is, CTRL-Y followed by the letter L should be coded as ^Y+L where the + denotes that the following character needs no CTRL or SHIFT.

18	File capture error: #Include had problems finding the named file.
19	Cannot find position in source file: Again to do with #include, another 'impossible' error reporting that it could not find the current position of the source file to remember it for after the #include.
20	Cannot have more than 4096 #defines: Each #define has to be checked for in each possible position in each source line: having too many of them slows the assembly noticeably. Although you can have up to 4096 #defines, there are also limits on the storage space for both the arguments (error 23); an average of eight characters for both arguments is recommended.
21	Run out of #define storage space (1): See error 20.
22	#define has no second argument: #define requires a **space** between the two parts of its argument to delimit it.
23	Run out of #define storage space (2): See errors 20 and 21 above; you have reached the limit of the storage space set aside for the second argument of #defines.
24	No strings in DC.W: Strings are only allowed as parts of BYTE, DC.B or STRING directives.
25	No strings in DC.L: Strings are only allowed as parts of BYTE, DC.B or STRING directives.

Table 82. Assembler errors (continued)

Error	Definition
26	Illegal external suffix: Only the suffixes .b, .B, .w, .W, .l, .L are legal after an external label in an external directive. If the suffix is left out then the default label size is used (as set by BYTES, WORDS or LONGS; default is WORDS).
27	Bad character in public line.
28	More than four characters in single quotes: This assembler uses double-quotes to surround string items and single-quotes to surround character constants. See *Section 4.3* on page 18.
29	Uneven single quotes: Single-quoted items must have a closing quote to delimit.
30	Sequential operator error: It does not allow arithmetic operators to be hard up against each other.
31	No lvalue in expression: An lvalue is the left-hand argument of an operator: a + b has 'a' as its lvalue. +b would cause this error.
32	Divide by zero: Attempt to divide a number by zero, in a numeric expression.
33	Ifs nested past 15 levels: Exceeded maximum number of nested #IF statements.
34	Spurious ELSE: An ELSE was discovered when no active IF was in force.
35	Spurious ENDIF: An ENDIF was discovered when no active IF was in force.
36	Only allowed inside Macros: A LOCAL directive was attempted outside a Macro definition.
37	No strings in word: See Error 24.
38	No string in long: See Error 25.
39	No REPEAT for this UNTIL: An UNTIL directive is found with no matching REPEAT directive.
40	Could not return to old place in source: Similar to Error 3 but generated by UNTIL instead of #include.
43	Syntax error in SKIP arguments: SKIP expects two numeric arguments, separated by a comma.
44	First SKIP argument is extern/relative; SKIP aborted: SKIP arguments must be known to the assembler absolutely. Extern or relative arguments are not allowed, although arithmetic is. If you need to move up to a new page, for example, use a new SEGMENT of the same class with a **page** align type.
45	Second SKIP argument is extern/relative; SKIP aborted: See Error 44.
46	Undefined label: This error happens when a reference is made to an undefined label.
47	Out of label space: A maximum of 1024 labels are allowed per module, and 10k is set aside to contain their names. If either of these limits is exceeded, this error results. Cut the module into two smaller ones; the assembly will happen twice as fast and you will only have to reassemble the half you have made changes to, speeding things up.
48	Label more than 30 characters: Labels longer than 30 characters are not allowed.
49	Label defined as PUBLIC twice: This warning occurs if the same label appears more than once in a PUBLIC statement. It is trapped because the second appearance may be a typographical error of a slightly different label you wanted declared as public.
50	Doubly defined label: This happens when a label is defined twice or more.

Table 82. Assembler errors (continued)

Error	Definition
51	Phase inconsistency (P1=X,P2=Y) 'label': Reports that the named label was allocated different values from pass-1 and pass-2, implies awful things. It's generally caused when for some reason the assembler has generated different lengths for the same instruction between pass-1 and pass-2. Sometimes if the assembler has problems identifying which addressing mode you wanted for a particular instruction because of typographical errors, or labels that are discovered to be undefined during the second pass it may give an error (see Error 54) and create no object code for that line. All labels after that line will then be allocated different values seeing as the object code is now that many bytes shorter, causing tons of Phase Inconsistency errors. Because this mass of Error 51s can sometimes hide the real cause of the error, a special assembler switch **/np** for 'no phase [errors]' can be used to switch them off. We strongly suggest that you don't always use **/np** on all your assemblies; only use it when you need it or you might miss critical phase errors.
52	Public symbol undefined: You defined a symbol in a PUBLIC directive that was not defined in the module.
53	Missing hex number: The assembler was led to expect a hex number but found one of zero length.
54	Cannot match addressing mode: This error is a catchall for the assembler if it cannot see anything wrong with your line but cannot match it to a known addressing mode either. There are two main causes of errors: significant ordering and numeric range errors. The significant ordering error is a simple typographical error: what should have been (val),y was coded as (valy, or whatever. All the components of the addressing mode are properly formed; it is just that the ordering is wrong. The numeric range errors can be harder to detect. For example, an 8-bit relative branch branching out of range would be trapped as an addressing mode error. To aid diagnostics of what went wrong the assembler dumps out its model of the line to the screen just before the error. Numerics are printed as a hex value followed by an attribute string: INTernal, EXTernal, ABSolute, RELative and .b, .w, .l. Significands are printed as the characters they represent, and strings are printed with their string. Numeric range errors are also trapped at the link stage (See *Section 5.1* on page 38).
55	Bad PSIG index: An 'impossible' error that could only occur through corruption of the .TAB file.
56	Un-recognized opcode: The Opcode (second field) could not be matched against any opcode names for this instruction set, nor could it be matched against any macro names or directives.
57	No closing quote: String must have closing double-quote before the end of the line.
58	No more than 12 numerics allowed on one line: There is a limit of 12 numeric units allowed on one line: this usually only matters on long DC.B-type directives where data tables are being defined. If it is a problem, simply cut the offending long line into two shorter lines.
59	Out of space for macro definition: The macro storage area (ca 64K) has overflowed. You must have some really big macros!
60	Too many macros attempted: There is a limit of 128 Macros allowed per source code module.
61	Mend only allowed in macro: MEND directive found with no matching MACRO directive.
62	No closing single quote: See error 29.
63	Bad ending: Another 'impossible' error, saying that the CR on the end of a source code line was missing.
64	Bad character in line: As each source code line is read into the assembler it is checked for non-ASCII characters (that is >128).
65	Parameter mismatch: The macro definition implies that there is a different number of parameters than there actually were in this calling line.
66	Currently unknown numeric type: An error in your Tabgen file, or a corrupted .TAB file: the numeric handler was asked to check a number against an undefined numeric type. Are you using the latest version of ASM.EXE and your .TAB file?

Table 82. Assembler errors (continued)

Error	Definition
67	Improper characters: Unusual characters have been spotted in the source file, of value >127.
68	Label used before its EXTERN definition: Labels must be declared EXTERNAL before use, preferably in a group at the top of the file.
69	Ambiguous label name: The label name in the single-quotes at the end of the error-line can be confused with a register name in this instruction set. Change the name.
70	Cannot have DS.X in segments containing code/data! (only for [void] segs!): DS.X does not produce any code; it simply advances the assembler's notional Program Counter. It cannot be used in the same segment as real 'code' or data.
71	Cannot have code in segments previously containing DS.X (only for non-void segs!): DS.X does not produce any code; it simply advances the assembler's notional Program Counter. It cannot be used in the same segment as real 'code' or data.
72	Constant too large for directive 'value': A DC.B cannot be given an argument >255, for example. Use LOW or OFFSET operators to truncate any wild arguments.
73	Could not find entry for segment in mapfile: This is for listings produced with '-fi' option. Complex include file structures and empty segments can sometimes throw the assembler off the track.
74	COD index only allowed on introduction: When you are using the multiple linker output file scheme, you can only specify the linker output file number for a particular class at the time of that class's introduction.
75	#LOAD before segment!: The #load has to be but at a given address! Before the first segment the assembler does not know what address to put it at! Shift the #load after a SEGMENT directive.
76	#LOAD before segment!: The assembler had problems finding the file you have named in a #LOAD directive.
77	All EQUs involving external args must be before first segment!
78	Cannot nest #includes > 5 levels
79	Could not find label list in mapfile: Happens with option '`-fi`' - Implies problem with the mapfile itself, or unsuccessful previous link, etc.
80	Could not find label in mapfile: As above. Is the Mapfile up to date with your edits? A label may be declared EXTERN, but never used.
81	Could not find label in mapfile: The date info has to be stored at a given address - before a SEGMENT there is no address information for the assembler to work on.
82	No string given on FCS line?: FCS is used for defining strings. Why is there no string on this line? Did you intend that? If so, use DC.B or BYTE.
83	Address not on WORD boundary: For 68000 and certain other genuine 16-bit, Opcodes must be on word boundary. This error occurs if you have assembled an instruction at an odd address. Your processor would crash!
84	Byte size label has value >255 (need WORDS?).
85	Word size label has value > 65535 (need LONGS?).
86	Over 250 macline pull: Internal error.
87	Run out of source file: Internal error.
88	TAB arguments incorrect: Must be in order num, num, num, num for example, TAB 8, 8, 12, 32.
89	Illegal suffix: An unknown suffix has been used with a label. Recognized suffixes are `.b`, `.w` and `.l`.
100	Label defined as NEAR and FAR

Table 82. Assembler errors (continued)

Error	Definition
101	Label defined as NEAR and INTERRUPT
102	Label defined as FAR and INTERRUPT
103	Label defined as NEAR twice
104	Label defined as FAR twice
105	Label defined as INTERRUPT twice

B.4 Linking errors

Table 83. Linking errors

Error	Definition
1	File list must be supplied
2, 3, 11, 13	Incomplete object file. Fatal error - the linker has identified that the given object file has been truncated. How are you for disk space?
4, 23	Size mismatch on EXTERN from F1 says .X, PUBLIC from F2 says .Y.. When declared PUBLIC in file F2 the label L was given size attribute Y. However, when you came to use it in file F1, the EXTERN statement named L as being of size .X - they did not tally. They must. Find out which is incorrect and alter it.
5	No info on start address of class 'class'. The first time the linker sees a class (remember it goes through the object files in the order given on the link command line), it must be given the full 'introduction' to the class, with start and stop addresses. See *Section 5.4* on page 40.
6	Too many secondary externals (32). Secondary externals ought to occur only rarely in your code - If you are using >32 then your structure has something seriously wrong with it. Hint - all your labels that are used all over the place, like constants, addresses of IO, and the like: make a module just for them, just containing EQUs and/or DS.Xs, all declared public. Any arithmetic needs doing more than once throughout your code, do it there, and declare the result with its own public label. Then refer to these PUBLICs using simple EXTERNs in each module.
7, 19	Corrupted object file. Disk nastiness. Reassemble. There may be an object code inconsistency: re-assemble all the files, and link again.
8	Public of same name as secondary EXTERN already exists! This error cannot be seen until link time. Rename one or the other.
9	Too many XREFs to link (12048)
10	Undefined EXTERN L (from F1)
12	Could not seek back in file 'F1'. Internal error. Should never occur.
14	Unexpected 7f. Disk nastiness. Reassemble.
15	Byte size exp >Offh 'value'. Needs looking at. If it is what you intended, either use the LOW operator to saw off upper bits, or change the size attribute.

Index

Symbols

Numerics

A

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

Revision history

Table 84. Document revision history

Date	Revision	Changes
01-Jul-2001	1	Initial release.
30-Jun-2005	2	Updated *Introduction* Updated *Getting started* installation procedure Added *Revision history*
05-June-2008	3	Reformatted document Added information on STM8 microcontroller support
20-Nov-2009	4	Added *Chapter 7: ABSLIST* Modified *Figure 1*, *Section 1.2*, *Table 1*, *Table 15*, and LI and FI options in *Section 4.7.2*

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER’S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com