



**Version 4.3**

***C Cross Compiler User's Guide  
for ST Microelectronics STM8***

**Copyright © COSMIC Software 1995, 2011**

**All Trademarks are the property of their respective owners**



# *Table of Contents*

---

## **Preface**

Organization of this Manual .....	1
-----------------------------------	---

## **Chapter 1**

### **Introduction**

Introduction.....	4
Document Conventions.....	4
Typewriter font.....	4
Italics .....	5
[ Brackets ] .....	5
Conventions.....	6
Command Line .....	6
Flags .....	6
Compiler Architecture .....	8
Predefined Symbol.....	9
Linking.....	9
Programming Support Utilities.....	9
Listings.....	10
Optimizations.....	10
Support for ROMable Code.....	11
Support for eeprom .....	12
Memory Models.....	12

## **Chapter 2**

### **Tutorial Introduction**

Acia.c, Example file.....	16
Default Compiler Operation .....	17
Compiling and Linking.....	19
Step 1: Compiling.....	19
Step 2: Assembler.....	20
Step 3: Linking .....	21
Step 4: Generating S-Records file .....	23
Linking Your Application.....	24
Generating Automatic Data Initialization.....	25
Specifying Command Line Options .....	28

## **Chapter 3**

### **Programming Environments**

Introduction.....	32
Modifying the Runtime Startup .....	34
Description of Runtime Startup Code .....	35

Initializing data in RAM.....	36
Memory Models for code smaller than 64K.....	39
Memory Models for code larger than 64K.....	39
Handling Large Code and Constants.....	41
Bit Variables.....	42
The const and volatile Type Qualifiers.....	43
Performing Input/Output in C.....	45
Referencing Absolute Addresses.....	46
Accessing Internal Registers.....	48
Placing Data Objects in The Bss Section.....	48
Placing Data Objects in Short Range Memory.....	49
Setting Short Range Size.....	49
Placing Data Objects in Long Range Memory.....	50
Placing Data Objects in the EEPROM Space.....	51
Redefining Sections.....	53
Inserting Inline Assembly Instructions.....	55
Inlining with pragmas.....	55
Inlining with <code>_asm</code> .....	56
Inlining Labels.....	58
Writing Interrupt Handlers.....	59
Placing Addresses in Interrupt Vectors.....	60
Inline Function.....	61
Interfacing C to Assembly Language.....	63
Register Usage.....	65
Heap Management Control with the C Compiler.....	66
Modifying The Heap Location.....	68
Data Representation.....	70

## Chapter 4

### Using The Compiler

Invoking the Compiler.....	74
Compiler Command Line Options.....	75
File Naming Conventions.....	80
Generating Listings.....	81
Generating an Error File.....	81
Return Status.....	81
Examples.....	81
C Library Support.....	82
How C Library Functions are Packaged.....	82
Inserting Assembler Code Directly.....	82
Linking Libraries with Your Program.....	82
Integer Library Functions.....	82
Common Input/Output Functions.....	83

Functions Implemented as Macros .....	83
Including Header Files .....	84
Descriptions of C Library Functions .....	85
Generate inline assembly code .....	86
Abort program execution .....	87
Find absolute value .....	88
Arccosine .....	89
Arcsine .....	90
Arctangent .....	91
Arctangent of y/x .....	92
Convert buffer to double .....	93
Convert buffer to integer .....	94
Convert buffer to long .....	95
Test or get the carry bit .....	96
Round to next higher integer .....	97
Verify the recorded checksum .....	98
Verify the recorded checksum .....	99
Verify the recorded checksum .....	100
Verify the recorded checksum .....	101
Cosine .....	102
Hyperbolic cosine .....	103
Divide with quotient and remainder .....	104
Copy a buffer to an eeprom buffer .....	105
Erase the eeprom space .....	106
Propagate fill character throughout eeprom buffer .....	107
Exit program execution .....	108
Exponential .....	109
Find double absolute value .....	110
Copy a moveable code segment in RAM .....	111
Round to next lower integer .....	112
Find double modulus .....	113
Extract fraction from exponent part .....	114
Get character from input stream .....	115
Get a text line from input stream .....	116
Test the interrupt mask bit .....	117
Test the interrupt line level .....	118
Test for alphabetic or numeric character .....	119
Test for alphabetic character .....	120
Test for control character .....	121
Test for digit .....	122
Test for graphic character .....	123
Test for lowercase character .....	124
Test for printing character .....	125

Test for punctuation character .....	126
Test for whitespace character .....	127
Test for uppercase character .....	128
Test for hexadecimal digit .....	129
Find long absolute value .....	130
Scale double exponent .....	131
Long divide with quotient and remainder .....	132
Natural logarithm .....	133
Common logarithm .....	134
Test for maximum .....	135
Scan buffer for character .....	136
Compare two buffers for lexical order .....	137
Copy one buffer to another .....	138
Copy one buffer to another .....	139
Propagate fill character throughout buffer .....	140
Test for minimum .....	141
Extract fraction and integer from double .....	142
Raise x to the y power .....	143
Output formatted arguments to stdout .....	144
Put a character to output stream .....	149
Put a text line to output stream .....	150
Generate pseudo-random number .....	151
Sin .....	152
Hyperbolic sine .....	153
Output arguments formatted to buffer .....	154
Real square root .....	155
Seed pseudo-random number generator .....	156
Concatenate strings .....	157
Scan string for first occurrence of character .....	158
Compare two strings for lexical order .....	159
Copy one string to another .....	160
Find the end of a span of characters in a set .....	161
Find length of a string .....	162
Concatenate strings of length n .....	163
Compare two n length strings for lexical order .....	164
Copy n length string .....	165
Find occurrence in string of character in set .....	166
Scan string for last occurrence of character .....	167
Find the end of a span of characters not in set .....	168
Scan string for first occurrence of string .....	169
Convert buffer to double .....	170
Convert buffer to long .....	171
Convert buffer to unsigned long .....	172

Tangent .....	173
Hyperbolic tangent .....	174
Convert character to lowercase if necessary .....	175
Convert character to uppercase if necessary .....	176

## Chapter 5

### Using The Assembler

Invoking castm8.....	178
Object File.....	181
Listings.....	181
Assembly Language Syntax.....	182
Instructions .....	183
Labels .....	183
Temporary Labels.....	184
Constants .....	185
Expressions.....	186
Macro Instructions.....	187
Conditional Directives.....	190
Sections.....	191
Bit Handling .....	192
Includes.....	193
Branch Optimization.....	194
Old Syntax .....	194
C Style Directives .....	195
Assembler Directives.....	195
Align the next instruction on a given boundary .....	196
Define the default base for numerical constants.....	197
Switch to the predefined .bsct section. ....	198
Turn listing of conditionally excluded code on or off. ....	199
Allocate constant(s) .....	200
Allocate constant block .....	201
Turn listing of debug directives on or off.....	202
Allocate variable(s) .....	203
Conditional assembly .....	204
Conditional assembly .....	205
Stop the assembly .....	206
End conditional assembly.....	207
End conditional assembly.....	208
End macro definition .....	209
End repeat section.....	210
Give a permanent value to a symbol .....	211
Assemble next byte at the next even address relative to the start of a section.....	212

Generate error message .....	213
Conditional assembly .....	214
Conditional assembly .....	215
Conditional assembly .....	216
Conditional assembly .....	217
Conditional assembly .....	218
Conditional assembly .....	219
Conditional assembly .....	220
Conditional assembly .....	221
Conditional assembly .....	222
Conditional assembly .....	223
Conditional assembly .....	224
Include text from another text file .....	225
Turn on listing during assembly .....	226
Give a text equivalent to a symbol .....	227
Create a new local block .....	228
Define a macro .....	229
Send a message out to STDOUT .....	231
Terminate a macro definition .....	232
Turn on or off listing of macro expansion.....	233
Turn off listing .....	234
Disable pagination in the listing file .....	235
Creates absolute symbols .....	236
Sets the location counter to an offset from the beginning of a section.....	237
Start a new page in the listing file .....	238
Specify the number of lines per pages in the listing file ..	239
Repeat a list of lines a number of times .....	240
Repeat a list of lines a number of times .....	241
Restore saved section .....	243
Terminate a repeat definition .....	244
Save section .....	245
Turn on or off section crossing.....	246
Define a new section .....	247
Give a resetable value to a symbol.....	249
Insert a number of blank lines before the next statement in the listing file.....	250
Place code into a section .....	251
Specify the number of spaces for a tab character in the listing file.....	252
Define default header .....	253
Declare bit symbol as being defined elsewhere .....	254
Declare a variable to be visible .....	255



Declare symbol as being defined elsewhere.....	256
--	-----

## Chapter 6

### Using The Linker

Introduction.....	259
Overview.....	260
Linker Command File Processing.....	262
Inserting comments in Linker commands .....	263
Linker Options .....	264
Global Command Line Options.....	265
Segment Control Options .....	267
Segment Grouping.....	271
Linking Files on the Command line .....	271
Example.....	272
Include Option .....	272
Example.....	273
Private Region Options.....	273
Symbol Definition Option .....	274
Reserve Space Option.....	275
Handle Dependencies .....	276
Section Relocation .....	277
Address Specification .....	277
Overlapping Control.....	277
Setting Bias and Offset .....	277
Setting the Bias.....	278
Setting the Offset.....	278
Using Default Placement.....	278
Bit Segment Handling .....	278
Linking Objects.....	280
Linking Library Objects.....	280
Library Order.....	281
Libraries Setup Search Paths.....	281
Automatic Data Initialization.....	283
Descriptor Format.....	283
Moveable Code.....	284
Manual Segment Initialization.....	286
Checksum Computation.....	287
DEFs and REFs.....	289
Special Topics.....	290
Private Name Regions .....	290
Renaming Symbols.....	291
Absolute Symbol Tables.....	294

Description of The Map File .....	295
Special Segments .debug and .info.....	296
Return Value.....	297
Linker Command Line Examples.....	298

## Chapter 7

### Debugging Support

Generating Debugging Information.....	302
Generating Line Number Information.....	302
Generating Data Object Information.....	302
The cprd Utility .....	304
Command Line Options .....	304
Examples .....	305
The clst utility.....	306
Command Line Options .....	306

## Chapter 8

### Programming Support

The chex Utility .....	310
Command Line Options .....	310
Return Status .....	312
Examples .....	312
The clabs Utility .....	314
Command Line Options .....	314
Return Status .....	315
Examples .....	315
The clib Utility.....	317
Command Line Options .....	317
Return Status .....	318
Examples .....	318
The cobj Utility.....	320
Command Line Options .....	320
Return Status .....	321
Examples .....	321
The Ctat Utility.....	322
Command Line Options .....	322
Return Status .....	323
Examples .....	323
The cvdwarf Utility .....	324
Command Line Options .....	324
Return Status .....	326
Examples .....	326

## **Chapter A**

### **Compiler Error Messages**

Parser (cpstm8) Error Messages .....	328
Code Generator (cgstm8) Error Messages.....	343
Assembler (castm8) Error Messages .....	344
Linker (clnk) Error Messages .....	347

## **Chapter B**

### **Modifying Compiler Operation**

The Configuration File.....	352
Changing the Default Options .....	354
Creating Your Own Options.....	354
Example .....	355

## **Chapter C**

### **STM8 Machine Library**

Update an int bitfield in near memory.....	358
Eeprom char bit field update .....	359
Write a char in eeprom .....	360
Write a long int in eeprom.....	361
Write a short int in eeprom.....	362
Move a structure in eeprom.....	363
Add float to float .....	364
Compare floats.....	365
Divide float by float.....	366
Add float to float in memory.....	367
Multiply float by float in memory.....	368
Subtract float from float in memory.....	369
Multiply float by float .....	370
Negate a float.....	371
Subtract float from float .....	372
Convert float to integer.....	373
Convert float into long integer .....	374
Compare a float in memory to zero .....	375
Get a long word from memory .....	376
Get a long word from memory .....	377
Get a word from far memory.....	378
Get a word from far memory.....	379
Get a word from far memory.....	380
Get a word from far memory.....	381
Get a word from far memory.....	382

Get a word from far memory.....	383
Quotient of integer division.....	384
Integer multiplication .....	385
Convert integer into float .....	386
Convert integer into long.....	387
Convert integer into long.....	388
Convert integer into long.....	389
Perform C switch statement on char.....	390
Perform C switch statement on long .....	391
Perform C switch statement on integer .....	392
Long integer addition .....	393
Long integer addition .....	394
Bitwise AND for long integers.....	395
Long integer compare.....	396
Quotient of long integer division .....	397
Long addition .....	398
Long addition .....	399
Long bitwise AND .....	400
Long shift left.....	401
Long multiplication in memory.....	402
Negate a long integer in memory .....	403
Long bitwise OR .....	404
Signed long shift right .....	405
Long subtraction.....	406
Long subtraction.....	407
Unsigned long shift right.....	408
Long bitwise exclusive OR .....	409
Long integer shift left.....	410
Remainder of long integer division .....	411
Multiply long integer by long integer .....	412
Negate a long integer.....	413
Bitwise OR with long integers .....	414
Long integer right shift.....	415
Long test against zero.....	416
Long integer subtraction.....	417
Long integer subtraction.....	418
Long integer compare with overflow .....	419
Convert long integer into float .....	420
Load memory into long register .....	421
Quotient of unsigned long integer division .....	422
Remainder of unsigned long integer division.....	423
Unsigned long integer shift right.....	424

Bitwise exclusive OR with long integers .....	425
Compare a long integer to zero .....	426
Far pointer addition .....	427
Far pointer addition .....	428
Put a long integer in memory .....	429
Put a long integer in memory .....	430
Put a word in far memory.....	431
Get a far pointer from far memory .....	432
Get a far pointer from far memory .....	433
Get a far pointer from far memory .....	434
Get a far pointer from far memory .....	435
Store long register in far memory.....	436
Store long register in memory .....	437
Quotient of signed char division.....	438
Quotient of signed char division.....	439
Remainder of signed char division .....	440
Remainder of signed char division .....	441
Multiply long integer by unsigned byte.....	442
Convert unsigned integer into float .....	443
Convert unsigned integer into long .....	444
Convert unsigned integer into long .....	445
Convert unsigned integer into long .....	446
Convert unsigned long integer into float.....	447
Multiply unsigned integers with long result.....	448
Multiply signed integers with long result.....	449
Store a far pointer into far memory .....	450
Copy a structure into another .....	451
Copy a structure in far memory.....	452
Copy a large structure in far memory.....	453
Copy a structure into another .....	454
Copy a large structure into another .....	455
Store a far pointer into far memory .....	456
Copy a structure into another .....	457
Copy a structure in far memory.....	458
Copy a large structure in far memory.....	459
Copy a structure into another .....	460
Copy a large structure into another .....	461

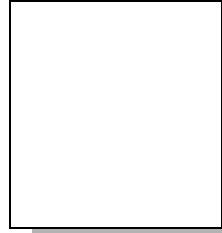
## Chapter D

### Compiler Passes

The cpstm8 Parser.....	464
Command Line Options .....	464

---

Warning Levels .....	469
Extra verifications .....	470
Return Status .....	471
Example.....	471
The cgstm8 Code Generator .....	472
Command Line Options .....	472
Return Status .....	474
Example.....	474
The costm8 Assembly Language Optimizer .....	475
Command Line Options .....	475
Disabling Optimization .....	476
Return Status .....	476
Example.....	476



# Preface

---

**T**he *Cross Compiler User's Guide for STM8* is a reference guide for programmers writing C programs for STM8 microcontroller environments. It provides an overview of how the cross compiler works, and explains how to compile, assemble, link and debug programs. It also describes the programming support utilities included with the cross compiler and provides tutorial and reference information to help you configure executable images to meet specific requirements. This manual assumes that you are familiar with your host operating system and with your specific target environment.

## Organization of this Manual

This manual is divided into eight chapters and four appendixes.

**Chapter 1**, “[Introduction](#)”, describes the basic organization of the C compiler and programming support utilities.

**Chapter 2**, “[Tutorial Introduction](#)”, is a series of examples that demonstrates how to compile, assemble and link a simple C program.

**Chapter 3**, “[Programming Environments](#)”, explains how to use the features of C for STM8 to meet the requirements of your particular application. It explains how to create a runtime startup for your application, and how to write C routines that perform special tasks such as: serial I/O, direct references to hardware addresses, interrupt handling, and assembly language calls.

**Chapter 4**, “[\*Using The Compiler\*](#)”, describes the compiler options. This chapter also describes the functions in the C runtime library.

**Chapter 5**, “[\*Using The Assembler\*](#)”, describes the STM8 assembler and its options. It explains the rules that your assembly language source must follow, and it documents all the directives supported by the assembler.

**Chapter 6**, “[\*Using The Linker\*](#)”, describes the linker and its options. This chapter describes in detail all the features of the linker and their use.

**Chapter 7**, “[\*Debugging Support\*](#)”, describes the support available for COSMIC's C source level cross debugger and for other debuggers or in-circuit emulators.

**Chapter 8**, “[\*Programming Support\*](#)”, describes the programming support utilities. Examples of how to use these utilities are also included.

**Appendix A**, “[\*Compiler Error Messages\*](#)”, is a list of compile time error messages that the C compiler may generate.

**Appendix B**, “[\*Modifying Compiler Operation\*](#)”, describes the “configuration file” that serves as default behaviour to the C compiler.

**Appendix C**, “[\*STM8 Machine Library\*](#)”, describes the assembly language routines that provide support for the C runtime library.

**Appendix D**, “[\*Compiler Passes\*](#)”, describes the specifics of the parser, code generator and assembly language optimizer and the command line options that each accepts.

This manual also contains an Index.



# Introduction

---

This chapter explains how the compiler operates. It also provides a basic understanding of the compiler architecture. This chapter includes the following sections:

- Introduction
- Document Conventions
- Compiler Architecture
- Predefined Symbol
- Linking
- Programming Support Utilities
- Listings
- Optimizations
- Support for ROMable Code
- Support for eeprom
- Memory Models

## Introduction

The C cross compiler targeting the STM8 microcontroller reads C source files, assembly language source files, and object code files, and produces an executable file. You can request listings that show your C source interspersed with the assembly language code and object code that the compiler generates. You can also request that the compiler generate an object module that contains debugging information that can be used by COSMIC's C source level cross debugger or by other debuggers or in-circuit emulators.

You begin compilation by invoking the **cxstm8** compiler driver with the specific options you need and the files to be compiled.

## Document Conventions

In this documentation set, we use a number of styles and typefaces to demonstrate the syntax of various commands and to show sample text you might type at a terminal or observe in a file. The following is a list of these conventions.

### *Typewriter font*

Used for user input/screen output. Typewriter (or courier) font is used in the text and in examples to represent what you might type at a terminal: command names, directives, switches, literal filenames, or any other text which must be typed exactly as shown. It is also used in other examples to represent what you might see on a screen or in a printed listing and to denote executables.

To distinguish it from other examples or listings, input from the user will appear in a shaded box throughout the text. Output to the terminal or to a file will appear in a line box.

For example, if you were instructed to type the compiler command that generates debugging information, it would appear as:

```
cxstm8 +debug acia.c
```

Typewriter font enclosed in a shaded box indicates that this line is entered by the user at the terminal.

If, however, the text included a partial listing of the file *acia.c* ‘an example of text from a file or from output to the terminal’ then typewriter font would still be used, but would be enclosed in a line box:

```
/* defines the ACIA as a structure */
struct acia {
    char status;
    char data;
} acia @0x6000;
```

### NOTE

*Due to the page width limitations of this manual, a single invocation line may be represented as two or more lines. You should, however, type the invocation as one line unless otherwise directed.*

### Italics

Used for value substitution. *Italic* type indicates categories of items for which you must substitute 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:

```
cxstm8 +debug file.c
```

In this example, `cxstm8 +debug file.c` is shown in typewriter font because it must be typed exactly as shown. Because the filename must be specified by the user, however, *file* is shown in italics.

### [ Brackets ]

Items enclosed in brackets are optional. For example, the line:

[ *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|stm8]
```

means that one file with the extension `.o` or `.stm8` may be specified, and the line:

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

means that additional files may be specified.

## Conventions

All the compiler utilities share the same optional arguments syntax. They are invoked by typing a command line.

## Command Line

A command line is generally composed of three major parts:

<pre>program_name [&lt;flags&gt;] &lt;files&gt;</pre>
---

where *<program\_name>* is the name of the program to run, *<flags>* an optional series of flags, and *<files>* a series of files. Each element of a command line is usually a string separated by whitespace from all the others.

## Flags

Flags are used to select options or specify parameters. Options are recognized by their first character, which is always a `-` or a `+`, followed by the name of the flag (usually a single letter). Some flags are simply *yes* or *no* indicators, but some must be followed by a value or some additional information. The value, if required, may be a character string, a single character, or an integer. The flags may be given in any order, and two or more may be combined in the same argument, so long as the second flag can't be mistaken for a value that goes with the previous one.

Each utility can display its version number, build date and host system by specifying the **-vers** option. The host name is ended by the **-F** sequence if a license is necessary for executing the utility.

It is possible for each utility to display a list of accepted options by specifying the **-help** option. Each option will be displayed alphabetically on a separate line with its name and a brief description. If an option requires additional information, then the type of information is indicated by one of the following code, displayed immediately after the option name:

Code	Type of information
*	character string
#	short integer
##	long integer
?	single character

If the code is immediately followed by the character '>', the option may be specified more than once with different values. In that case, the option name must be repeated for every specification.

For example, the options of the **chex** utility are:

```
chex [options] file
-a##      absolute file start address
-b##      address bias
-e##      entry point address
-f?       output format
-h        suppress header
+h*       specify header string
-m#       maximum data bytes per line
-n*>      output only named segments
-o*       output file name
-p        use paged address format
-pa       use paged address for data
-pl##     page numbers for linear mapping
-pn       use paged address in bank only
-pp       use paged address with mapping
-s        output increasing addresses
-w        output word addresses
-x*       exclude named segment
```

**chex** accepts the following distinct flags:

Flag	Function
<b>-a</b>	accept a long integer value
<b>-b</b>	accept a long integer value
<b>-e</b>	accept a long integer value
<b>-f</b>	accept a single character
<b>-h</b>	simply a flag indicator
<b>+h</b>	accept a character string
<b>-m</b>	accept a short integer value
<b>-n</b>	accept a character string and may be repeated
<b>-o</b>	accept a character string
<b>-p</b>	simply a flag indicator
<b>-pl</b>	accept a long integer value
<b>-pn</b>	simply a flag indicator
<b>-pp</b>	simply a flag indicator
<b>-s</b>	simply a flag indicator
<b>-w</b>	simply a flag indicator
<b>-x</b>	accept a character string and may be repeated

## Compiler Architecture

The C compiler consists of several programs that work together to translate your C source files to executable files and listings. **cxstm8** controls the operation of these programs automatically, using the options you specify, and runs the programs described below in the order listed:

**cpstm8** - the C preprocessor and language parser. **cpstm8** expands directives in your C source and parses the resulting text.

**cgstm8** - the code generator. *cgstm8* accepts the output of *cpstm8* and generates assembly language statements.

**costm8** - the assembly language optimizer. *costm8* optimizes the assembly language code that *cgstm8* generates.

**castm8** - the assembler. *castm8* converts the assembly language output of *costm8* to a relocatable object module.

## Predefined Symbol

The COSMIC compiler defines the `__CSMC__` preprocessor symbol. It expands to a numerical value whose each bit indicates if a specific option has been activated:

<b>bit 2</b>	set if unsigned char option specified ( <b>-pu</b> )
<b>bit 4</b>	set if reverse bitfield option specified ( <b>+rev</b> )
<b>bit 5</b>	set if no enum optimization specified ( <b>-pne</b> )

## Linking

**clnk** combines all the object modules that make up your program with the appropriate modules from the C library. You can also build your own libraries and have the linker select files from them as well. The linker generates an executable file which, after further processing with the *chex* utility, can be downloaded and run on your target system. If you specify debugging options when you invoke **cxstm8**, the compiler will generate a file that contains debugging information. You can then use the COSMIC's debugger to debug your code.

## Programming Support Utilities

Once object files are produced, you run **clnk** (the linker) to produce an executable image for your target system; you can use the programming support utilities to inspect the executable.

**chex** - absolute hex file generator. *chex* translates executable images produced by the linker into hexadecimal interchange formats, for use

with in-circuit emulators and PROM programmers. *chex* produces the following formats:

- Motorola S-record format
- standard Intel hex format

**clabs** - absolute listing utility. *clabs* translates relocatable listings produced by the assembler by replacing all relocatable information by absolute information. This utility must be used only after the linker.

**clib** - build and maintain object module libraries. *clib* allows you to collect related files into a single named library file for convenient storage. You use it to build and maintain object module libraries in standard library format.

**cobj** - object module inspector. *cobj* allows you to examine standard format executable and relocatable object files for symbol table information and to determine their size and configuration.

**cvdwarf** - ELF/DWARF format converter. *cvdwarf* allows you to convert a file produced by the linker into an **ELF/DWARF** format file.

## Listings

Several options for listings are available. If you request no listings, then error messages from the compiler are directed to your terminal, but no additional information is provided. Each error is labelled with the C source file name and line number where the error was detected.

If you request an assembly language and object code listing with interspersed C source, the compiler merges the C source as comments among the assembly language statements and lines of object code that it generates. Unless you specify otherwise, the error messages are still written to your terminal. Your listing is the listing output from the assembler.

## Optimizations

The C cross compiler performs a number of compile time and optimizations that help make your application smaller and faster:



- The compiler will perform arithmetic operations in 8-bit precision if the operands are 8-bit.
- The compiler eliminates unreachable code.
- Branch shortening logic chooses the smallest possible jump/branch instructions. Jumps to jumps and jumps over jumps are eliminated as well.
- Integer and float constant expressions are folded at compile time.
- Redundant load and store operations are removed.
- **enum** is large enough to represent all of its declared values, each of which is given a name. The names of **enum** values occupy the same space as type definitions, functions and object names. The compiler provides the ability to declare an **enum** using the smallest type char, int or long:
- The compiler performs multiplication by powers of two as faster shift instructions.
- An optimized switch statement produces combinations of tests and branches, jump tables for closely spaced case labels, a scan table for a small group of loosely spaced case labels, or a sorted table for an efficient search.
- The functions in the C library are packaged in three separate libraries; one of them is built without floating point support. If your application does not perform floating point calculations, you can decrease its size and increase its runtime efficiency by linking with the non-floating-point version of the modules needed.

## Support for ROMable Code

The compiler provides the following features to support ROMable code production. See Chapter 3 for more information.

- Referencing of absolute hardware addresses;
- Control of the STM8 interrupt system;

- Automatic data initialization;
- User configurable runtime startup file;
- Support for mixing C and assembly language code; and
- User configurable executable images suitable for direct input to a PROM programmer or for direct downloading to a target system.

## Support for eeprom

The compiler provides the following features to support **eeprom** handling:

- **@eeprom** type qualifier to describe a variable as an *eeprom* location. The compiler generates special sequences when the variable is modified.
- Library functions for erasure, initialization and copy of *eeprom* locations.

### NOTE

*The basic routine to program an eeprom byte is located in the library file **eeprom.s** and has been written using the default input/output address . This file must be modified if using a different base address.*

*These basic routines are not updating any watchdog, so applications enabling a watchdog must modify these routines to add watchdog updates in the wait loops.*

## Memory Models

The STM8 compiler supports several memory models allowing you to choose the best configuration for your processor and your application. You can choose to use 2-bytes addressing (applications smaller than 64k bytes) or 3-bytes addressing. You can also specify where variables are located in the memory space by default: inside or outside the **Short Range** (short addressing). For more information, please refer to Chapter 3.

For information on using the compiler, see [Chapter 4](#).

For information on using the assembler, see [Chapter 5](#).

For information on using the linker, see [Chapter 6](#).

For information on debugging support, see [Chapter 7](#).

For information on using the programming utilities, see [Chapter 8](#).

For information on the compiler passes, see [Appendix D](#).



# Tutorial Introduction

---

This chapter will demonstrate, step by step, how to compile, assemble and link the example program **acia.c**, which is included on your distribution media. Although this tutorial cannot show all the topics relevant to the COSMIC tools, it will demonstrate the basics of using the compiler for the most common applications.

In this tutorial you will find information on the following topics:

- [Default Compiler Operation](#)
- [Compiling and Linking](#)
- [Linking Your Application](#)
- [Generating Automatic Data Initialization](#)
- [Specifying Command Line Options](#)

## Acia.c, Example file

The following is a listing of *acia.c*. This C source file is copied during the installation of the compiler:

```

/*      STM8 EXAMPLE PROGRAM WITH INTERRUPT HANDLING
 *      Copyright (c) 2008 by COSMIC Software
 */
#include <iostm8.h>

#define SIZE64                /* buffer size */
#define TRDE0x80              /* transmit ready bit */

/*      Authorize interrupts.
 */
#define rim() _asm("rim")

/*      Some variables
 */
char buffer[SIZE];           /* reception buffer */
char *ptlec;                 /* read pointer */
char *ptecr;                 /* write pointer */

/*      Character reception.
 *      Loops until a character is received.
 */
char getch(void)
{
    char c;                  /* character to be returned */

    while (ptlec == ptecr) /* equal pointers => loop */
        ;
    c = *ptlec++;            /* get the received char */
    if (ptlec >= &buffer[SIZE]) /* put in in buffer */
        ptlec = buffer;
    return (c);
}

/*      Send a char to the SCI.
 */
void outch(char c)
{
    while (!(USART_SR & TRDE)) /* wait for READY */
        ;
    USART_DR = c;            /* send it */
}

```

```

/*    Character reception routine.
 *    This routine is called on interrupt.
 *    It puts the received char in the buffer.
 */
@interrupt void recept(void)
{
    USART_SR;                /* clear interrupt */
    *ptecr++ = USART_DR;      /* get the char */
    if (ptecr >= &buffer[SIZE]) /* put it in buffer */
        ptecr = buffer;
}

/*    Main program.
 *    Sets up the SCI and starts an infinite loop
 *    of receive transmit.
 */
void main(void)
{
    ptecr = ptlec = buffer; /* initialize pointers */
    USART_BRR1 = 0xc9;      /* parameter for baud rate */
    USART_CR1 = 0x00;        /* param. for word length */
    USART_CR2 = 0x2c;        /* parameters for interrupt */
    rim();                   /* authorize interrupts */
    for (;;)                 /* loop */
        outch(getch());      /* get and put a char */
}

```

## Default Compiler Operation

By default, the compiler compiles and assembles your program. You may then link object files using **clnk** to create an executable program.

The compiler supports several memory models, for applications smaller or larger than 64K, defining how the stack is used and where variables are allocated. A model option should always be specified on the command line; if nothing is specified, the compiler assumes the **+modsl** option (physical stack and globals in long range). See “[Memory Models for code smaller than 64K](#)” in **Chapter 3** and “[Memory Models for code larger than 64K](#)” in **Chapter 3** for more information.

As it processes the command line, **cxstm8** echoes the name of each input file to the standard output file (your terminal screen by default). You can change the amount of information the compiler sends to your terminal screen using command line options, as described later.

According to the options you will use, the following files, recognized by the COSMIC naming conventions, will be generated:

<b>file.s</b>	Assembler source module
<b>file.o</b>	Relocatable object module
<b>file.sm8</b>	input ( <i>e.g.</i> libraries) or output ( <i>e.g.</i> absolute executable) file for the linker



## Compiling and Linking

To compile and assemble *acia.c* using the short stack model, type:

```
cxstm8 +mods acia.c
```

The compiler writes the name of the input file it processes:

```
acia.c:
```

The result of the compilation process is an object module named *acia.o* produced by the assembler. We will, now, show you how to use the different components.

### Step 1: Compiling

The first step consists in compiling the C source file and producing an assembly language file named **acia.s**.

```
cxstm8 +mods -s acia.c
```

The **-s** option directs **cxstm8** to stop after having produced the assembly file *acia.s*. You can then edit this file with your favourite editor. You can also visualize it with the appropriate system command (*type*, *cat*, *more*,...). For example under MS/DOS you would type:

```
type acia.s
```

If you wish to get an interspersed C and assembly language file, you should type:

```
cxstm8 +mods -l acia.c
```

The **-l** option directs the compiler to produce an assembly language file with C source line interspersed in it. Please note that the C source lines are commented in the assembly language file: they start with **;**.

As you use the C compiler, you may find it useful to see the various actions taken by the compiler and to verify the options you selected.

The **-v** option, known as verbose mode, instructs the C compiler to display all of its actions. For example if you type:

```
cxstm8 +mods -v -s acia.c
```

the display will look like something similar to the following:

```
acia.c:
  cpstm8 -o \2.cx1 -i\cxstm8\hstm8 -u -hmods.h acia.c
  cgstm8 -o \2.cx2 \2.cx1
  costm8 -o acia.s \2.cx2
```

The compiler runs each pass:

<b>cpstm8</b>	the C parser
<b>cgstm8</b>	the assembly code generator
<b>costm8</b>	the optimizer

## Step 2: Assembler

The second step of the compilation is to assemble the code previously produced. The relocatable object file produced is *acia.o*.

```
cxstm8 acia.s
```

or

```
castm8 -i\cxstm8\hstm8 acia.s
```

if you want to use directly the macro cross assembler.

The cross assembler can provide, when necessary, listings, symbol table, cross reference and more. The following command will generate a listing file named *acia.ls* that will also contain a cross reference:

```
castm8 -c -l acia.s
```

For more information, see **Chapter 5**, “[Using The Assembler](#)”.

### Step 3: Linking

This step consists in linking relocatable files, also referred to as object modules, produced by the compiler or by the assembler (<files>.o) into an absolute executable file: **acia.sm8** in our example. Code and data sections will be located at absolute memory addresses. The linker is used with a command file (*acia.lkf* in this example).

An application that uses one or more object module(s) may require several sections (code, data, interrupt vectors, etc.,...) located at different addresses. Each object module contains several sections. The compiler creates the following sections:

Type	Description
<b>.text</b>	code (or program) section (e.g. <b>ROM</b> )
<b>.fconst</b>	large constant and literal data (e.g. <b>ROM</b> , see <b>@far</b> )
<b>.const</b>	constant and literal data (e.g. <b>ROM</b> )
<b>.data</b>	initialized data in <b>long addressing range</b> memory (see <b>@near</b> in chapter 3) (e.g. <b>RAM</b> )
<b>.bss</b>	all non initialized data in <b>long range</b> memory (e.g. <b>RAM</b> )
<b>.bsct</b>	initialized data in the first 256 bytes (see <b>@tiny</b> in chapter 3), also called <b>short range</b> or <b>short addressing range</b> .
<b>.ubsct</b>	non initialized data in the <i>short range</i>
<b>.fdata</b>	large variables ( <b>@far</b> )
<b>.eeprom</b>	any variable in eeprom ( <b>@eeprom</b> )
<b>.bit</b>	bit variables

In our example, and in the test file provided with the compiler, the *acia.lkf* file contains the following information:

```

line 1 # LINK COMMAND FILE FOR TEST PROGRAM
line 2 # Copyright (c) 2002 by COSMIC Software
line 3 #
line 4 +seg .text -b0xf000 -n.text# program start address
line 5 +seg .const -a .text # constants follow code

```

```

line 6 +seg .bsct -b0x0 -m0x100# short range start address
line 7 +seg .ubsct                # uninitialized short range
line 8 crt0.o                     # startup routine
line 9 acia.o                     # application program
line 10 \cx32\lib\libis.sm8       # C library (if needed)
line 11 \cx32\lib\libm.sm8        # machine library
line 12 +seg .vector -b0x8000 # vectors start address
line 13 vector.o                  # interrupt vectors

```

You can create your own link command file by modifying the one provided with the compiler.

Here is the explanation of the lines in *acia.lkf*:

**lines 1 to 3:** These are comment lines. Each line can include comments. They must be prefixed by the “#” character.

**line 4:** `+seg .text -b0xf000` creates a text (code) segment located at `f000` (hex address) named `.text`.

**line 5:** `+seg .const -n.text` creates a constant segment following the text segment.

**line 6:** `+seg .bsct -b0x0 -m0x100` creates a short range segment located at `0` (hex address) with a maximum size of 256 bytes.

**line 7:** `+seg .ubsct` creates an uninitialized short range segment located by default after the `.bsct` segment.

**line 8:** `crt0.o` runtime startup code. It will be located at `0xf000` (code segment)

**line 9:** `acia.o`, the file that constitutes your application. It follows the startup routine for code and data

**line 10:** `libis.sm8` the integer library to resolve references

**line 11:** `libm.sm8` the machine library to resolve references

**line 12:** `+seg .vector -b0x8000` creates a segment vector (const) segment located at `8000` (hex address)

**line 13:** `vectors.o` interrupt vectors file

By default and in our example, the *.bss* segment follows the *.data* segment.

The *crt0.o* file contains the runtime startup that performs the following operations:

- initialize the *bss*, if any
- initialize the stack pointer
- call *main()* or any other chosen entry point.

For more information, see “[Modifying the Runtime Startup](#)” in **Chapter 3**.

After you have modified the linker command file, you can link by typing:

```
clnk -o acia.sm8 acia.lkf
```

## Step 4: Generating S-Records file

Although *acia.sm8* is an executable image, it may not be in the correct format to be loaded on your target. Use the **chex** utility to translate the format produced by the linker into standard formats. To translate *acia.sm8* to *Motorola standard S-record* format:

```
chex acia.sm8 > acia.hex
```

or

```
chex -o acia.hex acia.sm8
```

**acia.hex** is now an executable image in *Motorola S-record* format and is ready to be loaded in your target system.

For more information, see “[The chex Utility](#)” in **Chapter 8**.

## Linking Your Application

You can create as many *text*, *data* and *bss* segments as your application requires. For example, assume we have one *bss*, two *data* and two *text* segments. Our link command file will look like:

```
+seg .text -b 0xf000 -n .text # program start address
+seg .const -a .text         # constant follow
+seg .data -b 0x100          # data start address
+seg .bss -b 0x200           # bss start address
+seg .bsct -b0x0 -m0x100    # zpage start address
+seg .ubsct -n iram          # uninitialized short range
crt.s.o                     # startup routine
acia.o                      # main program
module1.o                   # application program
+seg .text -b 0x2000         # start new text section
module2.o                   # application program
module3.o                   # application program
\cx\lib\libis.sm8           # C library (if needed)
\cx\lib\libm.sm8            # machine library
+seg .vector -b0x8000        # vectors start address
vector.o                    # interrupt vectors
#
# define these symbols if crt.s is used
#
#+def __endzp=@.ubsct        # end of uninitialized zpage
#+def __memory=@.bss         # symbol used by startup
```

In this example the linker will locate and merge *crt.s.o*, *acia.o* and *module1.o* in a *text* segment at 0xf000, a *data* segment at 0x100 and a *bss* segment, if needed at 0x200. *short range* variables will be located at 0x0. The rest of the application, *module2.o* and *module3.o* and the libraries will be located and merged in a new *text* segment at 0x2000 then the interrupt vectors file, *vector.o* in a *vector* segment at 0x8000. All constants will be located after the first *text* segment.

For more information about the linker, see **Chapter 6**, “[Using The Linker](#)”.

## Generating Automatic Data Initialization

Usually, in embedded applications, your program must reside in ROM.

This is not an issue when your application contains code and read-only data (such as string or const variables). All you have to do is burn a PROM with the correct values and plug it into your application board.

The problem comes up when your application uses initial data values that you have defined with initialized static data. These static data values must reside in RAM.

There are two types of static data initializations:

- 1) data that is explicitly initialized to a non-zero value:

```
char var1 = 25;
```

which is generated into the **.bsect** section and

- 2) data that is explicitly initialized to zero or left uninitialized:

```
char var2;
```

which is generated into the **.ubsect** section.

There is one exception to the above rules when you declare data that will be located in the **external memory**, using the **@near** type qualifier. In this case, the data is generated into the **.data** section if it is initialized or in the **.bss** section otherwise.

The first method to ensure that these values are correct consists in adding code in your application that reinitializes them from a copy that you have created and located in ROM, at each restart of the application.

The second method is to use the **crti.sm8** start-up file:

- 1) that defines a symbol that will force the linker to create a copy of the initialized RAM in ROM
- 2) and that will do the copy from ROM to RAM

The following link file demonstrates how to achieve automatic data initialization.

```
# demo.lkf: link command with automatic init
+seg .text -b 0xf000 -n .text # program start address
+seg .const -a .text          # constant follow
+seg .bsct -b 0x0 -m 0x100    # zpage start address
+seg .data -b0x100            # data start address
\cx32\lib\crtsi.sm8           # startup with auto-init
acia.o                        # main program
module1.o                     # module program
\cx32\lib\libis.sm8           # C library (if needed)
\cx32\lib\libm.sm8           # machine library
+seg .vector -b 0x8000        # vectors start address
vector.o                      # interrupt vectors
# define these symbols if crtsi is used
+def __endzp=@.ubsct          # end of uninitialized zpage
+def __memory=@.bss           # end of bss segment
```

In the above example, the *text* segment is located at address 0xf000, the *data* segment is located at address 0x100, immediately followed by the *bss* segment that contains uninitialized data. The initialized data in ROM will follow the descriptor created by the linker after the code segment.

In case of multiple code and data segments, a link command file could be:

```
# demoinit.lkf: link command with automatic init
+seg .text -b 0xf000 -n .text # program start address
+seg .const -a .text          # constant follow
+seg .bsct -b 0x0 -m 0x100    # zpage start address
+seg .data -b0x100            # data start address
\cx32\lib\crtsi.sm8           # startup with auto-init
acia.o                        # main program
module1.o                     # module program
+seg .text -b0xf800           # new code segment
module2.o                     # module program
module3.o                     # module program
\cx32\lib\libis.sm8           # C library (if needed)
\cx32\lib\libm.sm8           # machine library
+seg .vector -b 0x8000        # vectors start address
vector.o                      # interrupt vectors
# define these symbols if crtsi is used
+def __endzp=@.ubsct          # end of uninitialized zpage
+def __memory=@.bss           # end of bss segment
```



or

```
# demoinit.lkf: link command with automatic init
+seg .text -b 0xf000 -n .text # program start address
+seg .const -a .text          # constant follow
+seg .bsct -b 0x0 -m 0x100     # zpage start address
+seg .data -b 0x100           # data start address
\cx32\lib\crtsi.sm8          # startup with auto-init
acia.o                        # main program
module1.o                     # module program
+seg .text -b 0xf800 -it     # sets the section attribute
module2.o                     # module program
module3.o                     # module program
\cx32\lib\libis.sm8           # C library (if needed)
\cx32\lib\libm.sm8            # machine library
+seg .vector -b 0x8000         # vectors start address
vector.o                      # interrupt vectors
# define these symbols if crtsi is used
+def __endzp=@.ubsct          # end of uninitialized zpage
+def __memory=@.bss           # end of bss segment
```

In the first case, the initialized data will be located after the first code segment. In the second case, the ~~-it~~ option instructs the linker to locate the initialized data after the segment marked with this flag. The initialized data will be located after the second code segment located at address 0xf800.

For more information, see “[Initializing data in RAM](#)” in **Chapter 3** and “[Automatic Data Initialization](#)” in **Chapter 6**.

## Specifying Command Line Options

You specify command line options to **cxstm8** to control the compilation process.

To compile and get a relocatable file, type:

```
cxstm8 +mods acia.c
```

The file produced is *acia.o*.

The **-v** option instructs the compiler driver to echo the name and options of each program it calls. The **-l** option instructs the compiler driver to create a mixed listing of C code and assembly language code in the file *acia.ls*.

To perform the operations described above, enter the command:

```
cxstm8 +mods -v -l acia.c
```

When the compiler exits, the following files are left in your current directory:

- the C source file **acia.c**
- the C and assembly language listing **acia.ls**
- the object module **acia.o**

It is possible to locate listings and object files in specified directories if they are different from the current one, by using respectively the **-cl** and **-co** options:

```
cxstm8 +mods -cl\mylist -co\myobj -l acia.c
```

This command will compile the *acia.c* file, create a listing named *acia.ls* in the *\mylist* directory and an object file named *acia.o* in the *\myobj* directory.

**cxstm8** allows you to compile more than one file. The input files can be C source files or assembly source files. You can also mix all of these files.

If your application is composed with the following files: two C source files and one assembly source file, you would type:

```
cxstm8 +mods -v start.s acia.c getchar.c
```

This command will assemble the *start.s* file, and compile the two C source files.

See “[Compiler Command Line Options](#)” in **Chapter 4** for information on these and other command line options.



# Programming Environments

---

This chapter explains how to use the COSMIC program development system to perform special tasks required by various STM8 applications.

## Introduction

This chapter provides details about:

- [Modifying the Runtime Startup](#)
- [Initializing data in RAM](#)
- [Memory Models for code smaller than 64K](#)
- [Memory Models for code larger than 64K](#)
- [Handling Large Code and Constants](#)
- [Bit Variables](#)
- [The const and volatile Type Qualifiers](#)
- [Performing Input/Output in C](#)
- [Referencing Absolute Addresses](#)
- [Accessing Internal Registers](#)
- [Placing Data Objects in The Bss Section](#)
- [Placing Data Objects in Short Range Memory](#)
- [Placing Data Objects in Long Range Memory](#)
- [Placing Data Objects in the EEPROM Space](#)
- [Redefining Sections](#)
- [Inserting Inline Assembly Instructions](#)
- [Writing Interrupt Handlers](#)
- [Placing Addresses in Interrupt Vectors](#)
- [Inline Function](#)

- Interfacing C to Assembly Language
- Register Usage
- Heap Management Control with the C Compiler
- Data Representation

## Modifying the Runtime Startup

The runtime startup module performs many important functions to establish a runtime environment for C. The runtime startup file included with the standard distribution provides the following:

- Initialization of the **bss** section if any,
- ROM into RAM copy if required,
- Initialization of the stack pointer,
- *f\_main* or other program entry point call, and
- An exit sequence to return from the C environment. Most users must modify the exit sequence provided to meet the needs of their specific execution environment.

The following is a listing of the standard runtime startup file **crt0.asm** included on your distribution media. It does not perform automatic data initialization. The runtime startup file can be placed anywhere in memory. Usually, the startup will be “linked” with the **RESET** interrupt, and the startup file may be at any convenient location.

```
1 ; C STARTUP FOR STM8
2 ; WITHOUT ANY DATA INITIALIZATION
3 ; Copyright (c) 2006 by COSMIC Software
4 ;
5     xref  f_main, __stack
6     xdef  f_exit, __stext, f__stext
7 ;
8 ; startup routine from reset vector
9 ;
10    switch .text
11 __stext:
12 f__stext:
13 ;
14 ; initialize stack pointer
15 ;
16     ldw   x, #__stack ; stack pointer
17     ldw   sp, x        ; in place
18 ;
19 ; execute main() function
```



```

20 ; may be called by a 'jpf' instruction if no return
    expected
21 ;
22     callf f_main      ; execute main
23 f_exit:
24     jra    f_exit     ; and stay here
25 ;
26     end

```

## Description of Runtime Startup Code

**f\_main** is the entry point into the user C program.

**Line 15** resets the stack pointer.

**Line 22** calls *main()* in the user's C program.

**Lines 23 to 24** trap a return from *main()*. If your application must return to a monitor, for example, you must modify this line.

## Initializing data in RAM

If you have initialized static variables, which are located in **RAM**, you need to perform their initialization before you start your C program. The **clnk** linker will take care of that: it moves the initialized data segments after the **first** text segment, or the one you have selected with the **-it** option, and creates a descriptor giving the starting address, destination and size of each segment.

The table thus created and the copy of the **RAM** are located in **ROM** by the linker, and used to do the initialization. An example of how to do this is provided in the **crtsi.s** file, located in the headers sub-directory.

```
;      C STARTUP FOR STM8
;      WITH AUTOMATIC DATA INITIALISATION
;      Copyright (c) 2006 by COSMIC Software
;
;      xref f_main, __memory, __idesc__, __stack
;      xref.b c_x, c_y, __endzp
;      xdef f_exit, __stext, f__stext
;
;      start address of zpage
;
;      switch .ubsct
__suzp:
;
;      start address of bss
;
;      switch .bss
__sbss:
;
;      startup routine from reset vector
;
;      switch.text
__stext:
f__stext:
;
;      initialize stack pointer
;
;      ldw   x, #__stack ; stack pointer
;      ldw   sp, x       ; in place
;
;      setup initialized data
;
;      ldw   y, __idesc__ ; data start address
```

```

        ldw    x, #__idesc__+2    ; descriptor address
ibcl:
        ld     a, (x)             ; test flag byte
        jreq   zero              ; no more segment
        bcp    a, #$60            ; test for moveable code segment
        jreq   qbcl              ; yes, skip it
        ldw    c_x, x             ; save pointer
        ldw    x, (3, x)          ; move end address
        ldw    c_y, x             ; in memory
        ldw    x, c_x             ; reload pointer
        ldw    x, (1, x)          ; start address
dbcl:
        ld     a, (y)             ; transfer
        ld     (x), a             ; byte
        incw   x                  ; increment
        incw   y                  ; pointers
        cpw    y, c_y             ; last byte ?
        jrne   dbcl              ; no, loop again
        ldw    x, c_x             ; reload pointer
qbcl:
        addw   x, #5              ; next descriptor
        jra    ibcl              ; and loop
;
;   reset uninitialized data in zero page
;
zero:
        ldw    x, #__suzp        ; start of uninitialized zpage
        jra    loop              ; test segment end first
zbcl:
        ld     (x), a             ; clear byte
        incw   x                  ; next byte
loop:
        cpw    x, #__endzp       ; end of zpage
        jrne   zbcl              ; no, continue
;
;   reset uninitialized data in bss
;
        ldw    x, #__sbss        ; start address
        jra    ok                ; test segment end first
bbcl:
        ld     (x), a             ; clear byte
        incw   x                  ; next byte
ok:
        cpw    x, #__memory      ; compare end
        jrne   bbcl              ; not equal, continue
;

```

```

;      execute main() function
;      may be called by a 'jpf' instruction if no return
expected
;
;      callf f_main      ; execute main
f_exit:
;      jra   f_exit      ; and stay here
;
end

```

[\*crti.s\*](#) performs the same function as described with the [\*crt.s\*](#), but with one additional step. Lines (marked in bold) in [\*crti.s\*](#) include code to copy the contents of initialized static data, which has been placed in the text section by the linker, to the desired location in RAM.

The compiler is provided with several startup files implementing the automatic data initialization depending on the range of variables to be initialized and the range of the initialization table:

Startup	Initialize	From Table in
<b><a href="#">crti(0).s</a></b>	@near	@near
<b><a href="#">crtsx(0).s</a></b>	@near and @far	@near
<b><a href="#">crtisf(0).s</a></b>	@near	@far
<b><a href="#">crtxf(0).s</a></b>	@near and @far	@far

### NOTE

*When using a model for application smaller than 64K, you must use the specific startup (name ending with '0').*

For more information, see “[Generating Automatic Data Initialization](#)” in **Chapter 2** and “[Automatic Data Initialization](#)” in **Chapter 6**.

## Memory Models for code smaller than 64K

The STM8 compiler supports two memory models for application smaller than 64K, allowing you to choose the most efficient behavior depending on your processor configuration and your application. All these models handle code smaller than 64K and then function pointers are defaulted to **@near** pointers (2 bytes). Data pointers are defaulted to **@near** pointers (2 bytes). The supported models are:

- **Stack Short (mods0)** global variables are defaulted to *short range*. Any global object in long range will have to be accessed explicitly with the **@near** modifier unless accessed through a pointer.
- **Stack Long (mods10)** global variables are defaulted to *long range*. Any object in short range will have to be accessed explicitly with the **@tiny** modifier.

The choice of the appropriate model for a given application should be driven by the amount of globals compared with the available space in memory. **Short Range** variables are more efficient than **Long Range** variables but the **Short Range** size is limited to **256** bytes.

### NOTE

*When using a model for application smaller than 64K, you must link with the specific set of libraries and startup (names ending with '0').*

## Memory Models for code larger than 64K

The STM8 compiler supports two memory models for application larger than 64K, allowing you to choose the most efficient behavior depending on your processor configuration and your application. All these models allow the code to be larger than 64K and then function pointers are defaulted to **@far** pointers (3 bytes). Data pointers are defaulted to **@near** pointers (2 bytes) unless explicitly declared with the **@far** modifier. The supported models are:

- **Stack Short (mods)** global variables are defaulted to *short range*. Any global object in long range will have to be accessed explicitly with the **@near** modifier unless accessed through a pointer.

- **Stack Long (modsl)** global variables are also defaulted to *long range*. Any object in short range will have to be accessed explicitly with the **@tiny** modifier.

The choice of the appropriate model for a given application should be driven by the amount of globals compared with the available space in memory. **Short Range** variables are more efficient than **Long Range** variables but the **Short Range** size is limited to **256** bytes.

## Handling Large Code and Constants

The STM8 addresses more than 64K of code, dividing the total space in 64K large sections. The compiler allows by default functions to be as large as needed and to be allocated anywhere in the total space, regardless of any sections boundary crossing. The assembly name of functions allocated anywhere, implicitly declared as **@far** functions, is prefixed with the sequence “**f\_**”, the function being called by a **callf** instruction and ended with a **retf** instruction, while the name of functions declared explicitly as **@near** functions is prefixed with a single “**\_**”, the function being called by a **call** instruction and ended with a **ret** instruction. Such **@near** functions must be completely located inside the same section and called only by functions located in the same section. Unless trying to optimize the code space, the **@near** modifier should not be used. Such a difference in the function names allows the linker to check that **@far** and **@near** functions are called properly, any mixing being forbidden as the stack display would not be the same.

In order to allow large functions to cross section boundaries, the compiler has to use the far jump instruction **jpf** for any unconditional jump inside the function when the range is too large for a **jra** instruction. This makes the code larger even if such a function is not actually crossing a section boundary. It is possible to force the compiler using the **jp** instruction on any function by using the **-gnc** option. Such functions may still be **@far** or **@near** but cannot be larger than 64K and cannot be allocated across section. Such functions and **@near** functions are checked by the linker in order to verify this constraint.

Constants and literals are normally produced in the **.const** section which must be located in the first section. Large constants may be declared with an **@far** modifier. In such a case, they are produced in a **.fconst** section which may be located anywhere. Although the **far** space is used for code and constants, the compiler allows variables to be declared with the **@far** modifier. Such variables are allocated in a **.fdata** section which may be located anywhere.

## Bit Variables

The C compiler implements bit variables using the `_Bool` type name as defined by the new ANSI/ISO standard *C99* (also referenced as *C9X*). A `_Bool` variable is a boolean object which only contains the values *true* and *false*. They are implemented on single bits with value `1` for *true* and `0` for *false*. When assigning an expression to a `_Bool` variable, the compiler compares the expression against zero and assigns the boolean result to the boolean variable. So, any integer, real or pointer expression can be assigned to a boolean variable. It is not possible to declare arrays of booleans nor pointers to booleans, but booleans can be used as structure or union fields. Consecutive `_Bool` fields will be packed in bytes.

The compiler packs global `_Bool` variables in bytes using a bit section named `.bit`. Local `_Bool` variables are also packed in bytes regardless of the memory model. `_Bool` arguments are passed widened to a single byte.

```
_Bool in_range;
_Bool p_valid;
char *ptr;

in_range = (value >= 10) && (value <= 20);
p_valid = ptr;    /* p_valid is true if ptr not 0 */
if (p_valid && in_range)
    *ptr = value;
```



## The const and volatile Type Qualifiers

You can add the type qualifiers **const** and **volatile** to any base type or pointer type attribute.

*Volatile* types are useful for declaring data objects that appear to be in conventional storage but are actually represented in machine registers with special properties. You use the type qualifier **volatile** to declare memory mapped input/output control registers, shared data objects, and data objects accessed by signal handlers. The compiler will not optimize references to **volatile** data.

An expression that stores a value in a data object of **volatile** type stores the value immediately. An expression that accesses a value in a data object of **volatile** type obtains the stored value for each access. Your program will not reuse the value accessed earlier from a data object of **volatile** type.

### NOTE

*The volatile keyword must be used for any data object (variables) that can be modified outside of the normal flow of the function. Without the volatile keyword, all data objects are subject to normal redundant code removal optimizations. Volatile **MUST** be used for the following conditions:*

- 1) All data objects or variables associated with a memory mapped hardware register e.g. **volatile char DDRB @0x05;***
- 2) All **global** variable that can be modified (written to) by an interrupt service routine either directly or indirectly. e.g. a global variable used as a counter in an interrupt service routine.*

You use **const** to declare data objects whose stored values you do not intend to alter during execution of your program. You can therefore place data objects of **const** type in ROM or in write protected program segments. The cross compiler generates an error message if it encounters an expression that alters the value stored in a **const** data object.

If you declare a static data object of *const* type at either file level or at block level, you may specify its stored value by writing a data initializer. The compiler determines its stored value from its data initializer before program startup, and the stored value continues to exist unchanged until program termination. If you specify no data initializer, the stored value is zero. If you declare a data object of *const* type at argument level, you tell the compiler that your program will not alter the value stored in that argument in the related function. If you declare a data object of *const* type and dynamic lifetime at block level, you must specify its stored value by writing a data initializer. If you specify no data initializer, the stored value is undefined.

The *const* keyword implies the *@near* memory space to allow such a variable to be located in the code space. If a memory space modifier is explicitly given on a declaration using the *const* keyword, the compiler uses the given space instead of the default one, meaning that the object may not be located in the code space depending on the memory space given. In such a case, the *const* keyword still enforces the assignment checking.

You may specify *const* and *volatile* together, in either order. A *const volatile* data object could be a Read-only status register, or a status variable whose value may be set by another program.

Examples of data objects declared with type qualifiers are:

```
char * const x;      /* const pointer to char */
int * volatile y;    /* volatile pointer to int */
const float pi = 355.0 / 113.0; /* pi is never changed */
```

## Performing Input/Output in C

You perform input and output in C by using the C library functions *getchar*, *gets*, *printf*, *putchar*, *puts* and *sprintf*. They are described in chapter 4.

The C source code for these and all other C library functions is included with the distribution, so that you can modify them to meet your specific needs. Note that all input/output performed by C library functions is supported by underlying calls to *getchar* and *putchar*. These two functions provide access to all input/output library functions. The library is built in such a way so that you need only modify *getchar* and *putchar*, the rest of the library is independent of the runtime environment.

Function definitions for *getchar* and *putchar* are:

```
char getchar(void);  
char putchar(char c);
```

## Referencing Absolute Addresses

This C compiler allows you to read from and write to absolute addresses, and to assign an absolute address to a function entry point or to a data object. You can give a memory location a symbolic name and associated type, and use it as you would do with any C identifier. This feature is useful for accessing memory mapped I/O ports or for calling functions at known addresses in ROM.

References to absolute addresses have the general form `@<address>`, where `<address>` is a valid memory location in your environment. For example, to associate an I/O port at address `0x20` with the identifier name `ttystat`, write a definition of the form:

```
char MISCR1 @0x20;
```

where `@0x20` indicates an absolute address specification and not a data initializer. Since input/output on the STM8 architecture is memory mapped, performing I/O in this way is equivalent to writing in any given location in memory.

Such a declaration does not reserve any space in memory. The compiler still creates a label, using an *equate* definition, in order to reference the C object symbolically. This symbol is made *public* to allow external usage from any other file.

Individual bits can also be declared as `_Bool` objects by adding a bit number to the address using the syntax `@<address>:<bitnum>`, where `<address>` is a byte memory location and `<bitnum>` a bit number expressed by a constant value (or expression) between 0 and 7. For example, to define bit 3 of memory byte at `0x5001` as `PB3`:

```
_Bool PB3 @0x5001:3;
```

To use the I/O port in your application, write:

```
char c;  
c = MISCR1; /* to read from input port */  
MISCR1 = c; /* to write to output port */
```

Another solution is to use a **#define** directive with a cast to the type of the object being accessed, such as:

```
#define MISCR1 *(char *)0x20
```

which is both inelegant and confusing. The COSMIC implementation is more efficient and easier to use, at the cost of a slight loss in portability.

Note that COSMIC C does support the pointer and **#define** methods of implementing I/O access.

Another example of how to reference a direct memory address, defines a structure at absolute address **0x6000**:

```
struct acia
{
    char status;
    char data;
} acia @0x6000
```

Using this declaration, references to **acia.status** will refer to memory location **0x6000** and **acia.data** will refer to memory location **0x6001**. This is very useful if you are building your own custom I/O hardware that must reside at some location in the STM8 memory map.

## Accessing Internal Registers

All registers are declared in **io\*.h** files provided with the compiler, each one dedicated to a specific processor. One of these files should be included in each file using the input-output registers, for example by a:

```
#include <iostm8.h>
```

Note that the compiler will access to these registers as standard variables. In some case of reading or writing some “*int*” registers, you should declare an **union** (with two char and one int) instead of using directly the I/O register.

## Placing Data Objects in The Bss Section

The compiler automatically reserves space for uninitialized data object. All such data are placed in the **.bss** section. All initialized static data are placed in the **.data** section. The **bss** section is located, by default, after the **data** section by the linker.

The run-time startup files, *crt0.s* and *crti.s*, contain code which initializes the **bss** section space to zero.

The compiler provides a special option, **+nobss**, which forces uninitialized data to be explicitly located in the **.data** section. In such a case, these variables are considered as being explicitly initialized to zero.

## Placing Data Objects in Short Range Memory

The **Stack Short** model allocates global variables by default in *short range* memory. Such variables are located into the section **.bsct** if they are initialized, or in the section **.ubsc** otherwise. An external object name is published via a **xref.b** declaration at the assembly language level. A variable can be explicitly allocated in *short range* by using the **@tiny** modifier:

```
@tiny char c;
```

To place data objects into short range memory on a file basis, you can use the **#pragma** directive of the compiler:

```
#pragma space extern [] @tiny
```

instructs the compiler to place all data objects of storage class **extern** or **static** into short range memory for the current unit of compilation (usually a file). The section must end with a **#pragma space extern [] @near** to revert to the default compiler behaviour.

The compiler provides the *ctat* utility, which helps optimizing short range allocation by automatically extracting the list of all variables which can be efficiently moved to the short range area, by producing a header file which must be used to rebuild the application. For more information, see “[The Ctat Utility](#)” in **Chapter 8**.

### NOTE

*The code generator does not check for short range overflow.*

## Setting Short Range Size

You can set the size of the *short range* section of your object image at link time by specifying the following options on the linker command line:

```
+seg .bsct -m##
```

where **##** represents the size of the *short range* section in bytes. Note that the size of the *short range* section can never exceed **256 bytes**.

## Placing Data Objects in Long Range Memory

The **Stack Long** model allocates global variables by default in *long range* memory. Such variables will be located into the **.data** section if they are initialized, or in the **.bss** section otherwise. An external object name is published via a **xref** declaration at the assembly language level. A variable can be explicitly allocated in *Long Range* memory by using the **@near** modifier. The following declaration:

```
@near char ext;
```

instructs the compiler to locate the variable *ext* in the long range memory.

To place data objects into long range memory on a file basis, if not yet selected by the memory model, you can use the **#pragma** directive of the compiler:

```
#pragma space extern [] @near
```

instructs the compiler to place all data objects of storage class **extern** or **static** into *Long Range memory* for the current unit of compilation (usually a file).

The section must end with a **#pragma space extern [] @tiny** to revert to the default compiler behaviour.



## Placing Data Objects in the EEPROM Space

The compiler allows the use to define a variable as an **eeeprom** location, using the type qualifier **@eeeprom**. This causes the compiler to produce special code when such a variable is modified. When the compiler detects a write to an *eeeprom* location, it calls a machine library function which performs the actual write. An example of such a definition is:

```
@eeeprom char var;
```

To place all data objects from a file into *eeeprom*, you can use the **#pragma** directive of the compiler:

```
#pragma space extern [] @eeeprom @near
```

instructs the compiler to treat all *extern* and *static* data in the current file as *eeeprom* locations. The **@near** modifier is necessary because the *eeeprom* is located outside the *short range*.

The section must end with a **#pragma space extern [] @near**.

The compiler allocates *@eeeprom* variables in a separate section named **.eeeprom**, which will be located at link time.

The linker directive:

```
seg .eeeprom -b0x4000 -m2048  
var_eeeprom.o
```

will create a segment located at address **0x4000**, with a maximum size of 2048 bytes.

The library routines are optimized to take advantage of the four byte word programming feature wherever possible. The library routines are also using a few external symbols to locate the command registers in the I/O register space. When a standard header file provided with the compiler is included in at least one source file of the application, these symbols are automatically found. Otherwise, it is necessary to provide these symbols either by extra C declarations:

```
volatile char FLASH_IAPSR @0x505f;  
volatile char FLASH_CR2 @0x505b;  
volatile char FLASH_NCR2 @0x505c;
```

or by **+def** directives in the linker command file:

```
+def _FLASH_IAPSR=0x505f  
+def _FLASH_CR2=0x505b  
+def _FLASH_NCR2=0x505c
```

The addresses provided must match the proper register addresses of the target device.

The **STM8L** devices do not implement the **FLASH\_NCR2** register. In order to use the eeprom support for these devices, you must link with the extra library **libl.sm8** placed **before** the standard **libm.sm8**, and you must not declare the matching symbol if the standard I/O file is not included in the application.

#### NOTE

*The code generator cannot check if data address will be eeprom addresses after linkage.*

## Redefining Sections

The compiler uses by default predefined sections to output the various component of a C program. The default sections are:

Section	Description
<b>.text</b>	executable code
<b>.const</b>	text string and constants
<b>.fconst</b>	large constant variables ( <b>@far</b> )
<b>.data</b>	initialized variables ( <b>@near</b> )
<b>.bss</b>	uninitialized variables ( <b>@near</b> )
<b>.bsct</b>	initialized variables in short range ( <b>@tiny</b> by default)
<b>.ubsct</b>	uninitialized variables in short range ( <b>@tiny</b> by default)
<b>.fddata</b>	large variables ( <b>@far</b> )
<b>.eeprom</b>	any variable in eeprom ( <b>@eeprom</b> )
<b>.bit</b>	bit variables

It is possible to redirect any of these components to any user defined section by using the following pragma definition:

```
#pragma section <attribute> <qualified_name>
```

where <attribute> is either **empty** or one of the following sequences:

```
const
_Bool
@tiny
@near
@far
@eeprom
```

and <qualified\_name> is a section name enclosed as follows:

```
(name) - parenthesis indicating a code section
[name] - square brackets indicating uninitialized data
{name} - curly braces indicating initialized data
```

A section name is a plain C identifier which *does not* begin with a dot character, and which is no longer than **13** characters. The compiler will prefix automatically the section name with a dot character when passing this information to the assembler. It is possible to switch back to the default sections by omitting the section name in the *<qualified\_name>* sequence.

Each pragma directive starts redirecting the selected component from the next declarations. Redefining the *bss* section forces the compiler to produce the memory definitions for all the previous *bss* declarations before to switch to the new section.

The following directives:

```
#pragma section (code)
#pragma section const {string}
#pragma section @near [udata]
#pragma section @near {idata}
#pragma section @tiny [uzpage]
#pragma section @tiny {izpage}
#pragma section @eeprom @near {e2prom}
#pragma section _Bool {bdata}
```

redefine the default sections (or the previous one) as following:

- executable code is redirected to section **.code**
- strings and constants are redirected to section **.string**
- uninitialized variables are redirected to section **.udata**
- initialized data are redirected to section **.idata**
- uninitialized zpage variables are redirected to section **.uzpage**
- initialized zpage variables are redirected to section **.izpage**
- eeprom variables are redirected to section **.e2prom**
- bit variables are redirected to section **.bdata**

Note that **{name}** and **[name]** are equivalent for constant and eeprom sections as they are all considered as initialized.

The following directive:

```
#pragma section ()
```

switches back the code section to the default section **.text**.

## Inserting Inline Assembly Instructions

The compiler features two ways to insert assembly instructions in a C file. The first method uses **#pragma** directives to enclose assembly instructions. The second method uses a special function call to insert assembly instructions. The first one is more convenient for large sequences but does not provide any connection with C object. The second one is more convenient to interface with C objects but is more limited regarding the code length.

### Inlining with pragmas

The compiler accepts the following pragma sequences to start and finish assembly instruction blocks:

Directive	Description
<b>#pragma asm</b>	start assembler block
<b>#pragma endasm</b>	end assembler block

The compiler also accepts shorter sequences with the same meaning:

Directive	Description
<b>#asm</b>	start assembler block
<b>#endasm</b>	end assembler block

Such an assembler block may be located anywhere, inside or outside a function. Outside a function, it behaves syntactically as a declaration. This means that such an assembler block cannot split a C declaration somewhere in the middle. Inside a function, it behaves syntactically as one C instruction. This means that there is no trailing semicolon at the end, and no need for enclosing braces. It also means that such an assembler block cannot split a C instruction or expression somewhere in the middle.

The following example shows a correct syntax:

```
#pragma asm
    xref asmvar
#pragma endasm

extern char test;

void func(void)
{
    if (test)
#asm      /* no need for { */
    scf    ; set carry bit
    rlc asmvar; access assembler variable
#endasm
    else
        test = 1;
}
```

### Inlining with `_asm`

The `_asm()` function inserts inline assembly code in your C program. The syntax is:

```
_asm("string constant", arguments...);
```

The “*string constant*” argument is the assembly code you want embedded in your C program. “*arguments*” follow the standard C rules for passing arguments. The string you specify follows standard C rules. For example, carriage returns can be denoted by the ‘`\n`’ character.

For example, to produce the following assembly sequence:

```
ldw x,sp
callf f_main
```

you would write

```
_asm("ldw x,sp\n callf f_main\n");
```

The ‘`\n`’ character is used to separate the instructions when writing multiple instructions in the same line.

**NOTE**

The argument string **must** be **shorter** than **255** characters. If you wish to insert longer assembly code strings you will have to split your input among consecutive calls to `_asm()`.

`_asm()` does not perform any checks on its argument string. Only the assembler can detect errors in code passed as argument to an `_asm()` call.

`_asm()` can be used in expressions, if the code produced by `_asm` complies with the rules for function returns. For example:

```
var = _asm("sra a\n rlc a\n rlc a\n", var);
```

allows to rotate the variable `var` passed as argument in the **a** register, and store the result in the same variable. The variable `var` is supposed to be declared as a `char`, and is loaded in the **a** register because it is considered as a first argument. The result is expected in the **a** register in order to comply with the return register convention, as described below.

**NOTE**

With both methods, the assembler source is added as is to the code during the compilation. The optimizer **does not** modify the specified instructions, unless the **-a** option is specified on the code generator. The assembler input can use lowercase or uppercase mnemonics, and may include assembler comments.

By default, `_asm()` is returning an **int** as any undeclared function. To avoid the need of several definitions (usually conflictuous) when `_asm()` is used with different return types, the compiler implements a special behaviour when a cast is applied to `_asm()`. In such a case, the cast is considered to define the return type of `_asm()` instead of asking for a type conversion. There is no need for any prototype for the `_asm()` function as the parser verifies that the first argument is a string constant.

## Inlining Labels

When labels are necessary in the inlined assembly code, the compiler provides a special syntax allowing local labels to be created and handled without interaction with other labels and the optimizer. The sequence **\$N** in the assembly source is replaced by a new label name while the sequence **\$L** is replaced by the label name created by the last **\$N**. Using this syntax, a simple wait loop may be entered as follow:

```
#asm
    ld    a,#7
$N:
    dec   a
    jrne  $L    ; loop on the previous label
#endasm
```



## Writing Interrupt Handlers

A function declared with the type qualifier **@interrupt** is suitable for direct connection to an interrupt (hardware or software). **@interrupt** functions may not return a value. **@interrupt** functions are allowed to have arguments, although hardware generated interrupts are not likely to supply anything meaningful.

When you define an **@interrupt** function, the compiler uses the “**iret**” instruction for the return sequence, and saves, *if necessary*, the memory bytes used by the compiler for its internal usage. Those areas are **c\_x** (3 bytes), **c\_y** (3 bytes) and **c\_lreg** (4 bytes). Those bytes will be saved and restored if the interrupt function uses them directly. If the interrupt function does not use these areas directly, but calls another C function, the **c\_x** and **c\_y** areas will be automatically saved and restored, unless using the type qualifier **@nosvf** on the interrupt function definition. This qualifier can be used when the called functions are known not using those areas, but the compiler does not perform any verification. The **c\_lreg** area is not saved implicitly in such a case, in order to keep the interrupt function as efficient as possible. If any function called by the interrupt function uses **longs** or **floats**, the **c\_lreg** area can be saved by using the type qualifier **@svlreg** on the interrupt function definition. Whatever the model used is, these copies are made directly on the stack.

You define an **@interrupt** function by using the type qualifier **@interrupt** to qualify the type returned by the function you declare. An example of such a definition is:

```
@interrupt void it_handler(void)
{
    ...
}
```

### NOTE

The **@interrupt** modifier is an extension to the ANSI standard.

When an interrupt function uses a **div** or **divw** instruction, the compiler produces a special entry sequence protecting these instructions from a silicon bug (cf. STM8 errata sheet available on the ST web site). The compiler detects if a function uses directly these instructions or calls a

machine library routine using these instructions. If an interrupt function calls another C function, the compiler produces this sequence unconditionally unless the interrupt function has been declared with the type qualifier **@nopr** and no direct usage has been detected.

This automatic process can be disabled by compiling with option **-gdp**. In such a case, the programmer must add protecting sequences by inline assembly code wherever it is necessary.

## Placing Addresses in Interrupt Vectors

The compiler implements the special modifier **@vector** in order to allow the interrupt table to be declared directly in C. Each entry must be declared as a function pointer and initialized with a function name. The compiler will produce instead of an address constant, a 4 byte entry containing the special opcode **0x82** followed by the 24 bit address of the interrupt function to be reached.

Refer to the *vector.c* file provided with the compiler example for a more accurate implementation description.

A small C construct would be:

```
extern void handler1(), handler2(), handler3();
void (* const @vector vectab[]) () =
{
    handler1,
    handler2,
    handler3,
};
```

where *handler1* and so forth are interrupt handlers, which can be located anywhere in the code space, meaning **@near** or **@far** functions. Then, in the linker command file, include the following options on the directive line:

```
+seg .const -b0x8000 vector.o
```

where *vector.o* is the file which contains the vector table. This file is provided in the compiler package.

## Inline Function

The compiler is able to inline a function body instead of producing a function call. This feature allows the program to run faster but produces a larger code. A function to be inlined has to be defined with the **@inline** modifier. Such a function is kept by the compiler and does not produced any code yet. Each time this function is called in the same source file, the call is replaced by the full body of the inlined function. Because inlined functions are in fact local to a source file, they should be defined in a header file if they have to be used by several source files. To allow the arguments to be passed properly, inlined functions must be defined with prototypes.

### NOTE

*Inline functions cannot declare static local variables and cannot call themselves either directly or indirectly.*

The compiler allows access to specific instructions or features of the STM8 processor, using **@inline** functions. Such functions shall be declared as external functions with the **@inline** modifier. The compiler recognizes three predefined functions when explicitly declared as follows:

```
@inline char carry(void);
@inline char irq(void);
@inline char imask(void);
```

<b>carry</b>	the <b>carry</b> function is used to test or get the carry bit from the condition register. If the <b>carry</b> function is used in a test, the compiler produces a <b>jnrc</b> or <b>jrc</b> instruction. If the <b>carry</b> function is used in any other expression, the compiler produces a code sequence setting the <b>a</b> register to 0 or 1 depending on the carry bit value.
<b>irq</b>	the <b>irq</b> function is used to test the interrupt line level using the <b>jrih</b> or <b>jril</b> instruction. The <b>irq</b> function can be used only in a test
<b>imask</b>	the <b>imask</b> function is used to test the interrupt mask bit in the condition register using the <b>jrm</b> or <b>jrnrm</b> instruction. The <b>imask</b> function can be used only in a test.

These functions are predeclared in the *processor.h* header file. A full description with examples is provided in Chapter 4.

Any other function declared as an *@inline* will be translated into a call to a user provided macro. The macro name is obtained by prefixing the *@inline* function name with the ‘\_’ character. Arguments are allowed but should be restricted to variable references. Each reference is translated into the proper assembler expression (same translation as applied by the compiler) and then passed to the macro as a quoted text string.

*@inline* functions may use the registers **a** and/or **x**, but the compiler can not check their use and will not save them. To save the registers before they are used by *@inline* functions, you must add the *@usea* and/or the *@usex* modifiers.

For example:

```
@inline @usea lsub();
```

tells the compiler that *lsub()* uses the register **a**, so that the compiler will save it. If both registers are used, you must specify both modifiers.

## Interfacing C to Assembly Language

The C cross compiler translates C programs into assembly language according to the specifications described in this section.

You may write external identifiers in both uppercase and lowercase. The compiler prepends an underscore ‘\_’ character to each identifier. If the identifier is the name of an **@far** function, the compiler prepends a ‘f’ character to the extra underscore.

The compiler places function code in the **.text** section. Function code is not to be altered or read as data. External function names are published via **xdef** declarations.

Compiler literals, containing text strings, float and long constants, and switch tables, are generated by default into the **.const** section. An option on the code generator allows such constants to be produced in the **.text** section. In any case, these literals must be linked in the **@near** space.

The compiler generates initialized data declared with the **@near** modifier into the **.data** section. Such external data names are published via **xref** declarations. Data you declare to be of “const” type by adding the type qualifier *const* to its base type is generated by default into the **.const** section. Const variables declared with the **@far** modifier are published in the **.fconst** section. Initialized data declared with the **@tiny** space modifier will be generated into the **.bsct** section. Such external data names are published via **xref.b** declarations. Uninitialized data are normally generated into the **.bss** section for **@near** variables or the **.ubsct** section for **@tiny** variables, unless forced to the **.data** or **.bsct** section by the compiler option **+nobss**. Variables declared with the **@far** modifier are published in the **.fdata** section. **\_Bool** data is generated in the **.bit** section and external names are published via **xbit.b** declarations.

Section	Declaration	Reference
<b>.bsct</b>	@tiny char i =2;	xdef
<b>.ubsct</b>	@tiny char i;	xdef

Section	Declaration	Reference
<b>.data</b>	int init = 1	xdef
<b>.bss</b>	int uninit	xdef
<b>.text</b>	char putchar(c);	xdef
<b>.const</b>	const char c = 1;	xdef
<b>.fconst</b>	@far const char c = 1;	xdef
<b>.fdata</b>	@far char i;	xdef
<b>.bit</b>	_Bool Pb3;	xdef
Any of above	extern int out;	xref, xbit

Function calls are performed according to the following:

- 1) Arguments are evaluated from right to left. The first argument is stored in the **a** register if it is a char or a **@tiny** pointer, or in the **x** register if its type is *short*, *int* or **@near** pointer, or the **x** register and the memory location **c\_x** if its type is **@far** pointer, and if the function does not return a structure larger than 2 bytes. Other arguments are pushed to the stack. If the two first arguments are of type *char* or *unsigned char*, they are both stored in registers, the first in the **xh** register, and the second in the **xl** register as if they were considered as an *int*.
- 2) The function is called via a **callf func** instruction.
- 3) Stacked arguments are cleaned out.

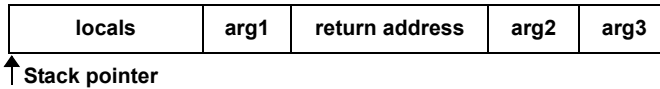
## Register Usage

Except for the return value, the registers **a**, **x**, **y** and the condition codes are undefined on return from a function call. The return value is in **a** if it is of type char, **@tiny** pointer or a one byte structure, **x** if it is of type short, integer, **@near** pointer or a two byte structure. The return value is in the memory located at symbol **c\_lreg** if it is of type long or float. The return value is in the memory location **c\_x** (upper byte) and the **x** register (lower word) if it is of type **@far** pointer.

The first argument may be hold in register, and will be stored at the function entry. Such a function declaration:

```
int func(int arg1, int arg2, int arg3)
```

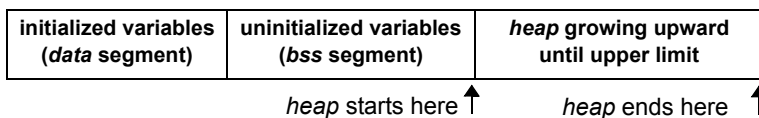
will create the following memory area:



## Heap Management Control with the C Compiler

The name **heap** designates a memory area in which are allocated and deallocated memory blocks for temporary usage. A memory block is allocated with the *malloc()* function, and is released with the *free()* function. The *malloc()* function returns a pointer to the allocated area which can be used until it is released by the *free()* function. Note that the *free()* function has to be called with the pointer returned by *malloc*. The *heap* allocation differs from a local variable allocation because its life is not limited to the life of the function performing the allocation.

In an embedded application, the *malloc-free* mechanism is available and automatically set up by the compiler environment and the library. But it is possible to control externally the *heap* size and location. The default compiler behaviour is to create a data area containing application variables and *heap* in the following way:



The heap start is the *bss* end, and is equal to the **\_\_startmem** symbol defined by the linker with an appropriate **+def** directive. The heap end is equal to the **\_\_endmem** symbol defined by the linker with an appropriate **+def** directive. A typical heap reservation may be implemented in the linker command file by using the **+spc** directive:

```
+def __startmem=@.bss    # heap start symbol
+spc .bss=0x400          # reserve 1K space
+def __endmem=@.bss      # heap end
```

The *heap* management functions maintain a global pointer named *heap pointer*, or simply **HP**, pointing to the *heap top*, and a linked list of memory blocks, free or allocated, in the area between the *heap start* and the *heap top*. In order to be able to easily modify the heap implementation, the heap management functions use a dedicated function to move the heap pointer whenever necessary. The heap pointer is initialized to the heap start: the heap is initially empty. When *malloc* needs some memory and no space is available in the free list, it calls this dedicated function named **\_sbreak** to move the heap pointer upwards if possible.



`_sbreak` will return a NULL pointer if this move is not possible (usually this is because the heap would overlap the stack). Therefore it is possible to change the heap default location by rewriting the `_sbreak` function.

The default `_sbreak` function provided by the library is as follows:

```
/*      SET SYSTEM BREAK
 */
void *sbreak(int size)
{
    extern char _startmem, _endmem;
    static char *_brk = NULL; /* memory break */
    char *obrk;

    if (!_brk) /* initialize on first call */
        _brk = &_startmem;
    obrk = _brk; /* old top */
    _brk += size; /* new top */
    if (&_endmem <= _brk || _brk < &_startmem)
    {
        /* check boundaries */
        _brk = obrk; /* restore old top */
        return (NULL); /* return NULL pointer */
    }
    return (obrk); /* return new area start */
}
```

If the new top is outside the authorized limits defined by the address of the two objects `_startmem` and `_endmem`, the function returns a NULL pointer, otherwise, it returns the start of the new allocated area. Note that the top variable `_brk` is a static variable initialized to zero (NULL pointer). It is set to the heap start on the first call. It is also possible to initialize it directly within the declaration, but in this case, we create an initialized variable in the **data** segment which needs to be initialized by the startup. The current code avoids such a requirement by initializing the variable to zero (in the **bss** segment), which is simply done by the standard startup sequence.

## Modifying The Heap Location

It is easy to modify the `_sbreak` function in order to handle the heap in a standard C array, which will be part of the application variables.

The heap area is declared as an array of **char** simply named *heap*. The algorithm is mainly the same, and once the new top is computed, it is compared with the array limits. Note that the array is declared as a static local variable. It is possible to have it declared as a static global variable. If you want it to be global, be careful on the selected name. You should start it with a `'_'` character to avoid any conflict with the application variables.

The modified `_sbreak` function using an array is as follows:

```
/*      SET SYSTEM BREAK IN AN ARRAY
*/
#define HSIZE 800/* heap size */

void *sbreak(int size)
{
    static char *_brk = NULL; /* memory break */
    static char heap[HSIZE]; /* heap area */
    char *obrk;

    if (!_brk)                /* initialize on first call */
        _brk = heap;
    obrk = _brk;              /* old top */
    _brk += size;             /* new top */
    if (&heap[HSIZE] <= _brk || _brk < heap)
    {
        /* check boundaries */
        _brk = obrk;         /* restore old top */
        return (NULL);       /* return NULL pointer */
    }
    return (obrk);           /* return new area start */
}
```

If you need to place the heap array at a specific location, you need to locate this module at a specific address using the linker options. In the above example, the heap array will be located in the `.bss` segment, thus, complicating the startup code which would need to zero two `bss` sections instead of one. Compiling this function, with the **+nobss** option, will force allocation of the heap, in the data segment and you can locate it easily with linker directives as:

```
+seg .data -b 0x2000    # heap start
sbreak.o               # sbreak function
```

It is also possible to handle the heap area outside of any C object, just by redefining the heap start and end values using the linker **+def** directives:

```
+def __startmem=0x2000 # heap start
+def __endmem=0x3000   # heap end
```

### NOTE

*Since the initial content of the area may be undefined, the **-ib** option should be specified to exclude the segment in the automatic RAM initialization.*

Note that it is possible to use this `_sbreak` function as a `malloc` equivalent function with some restrictions. The `malloc` function should be used when the allocated memory has to be released, or if the application has no idea about the total amount of space needed. If memory can be allocated and never released, the free mechanism is not necessary, nor the linked list of memory blocks built by `malloc`. In that case, simply rename the `_sbreak` function as `malloc`, regardless of its implementation, and you will get a very efficient and compact `malloc` mechanism. You may do the renaming in the function itself, which needs to be recompiled, or by using a `#define` at C level, or by renaming the function at link time with a **+def** directive such as:

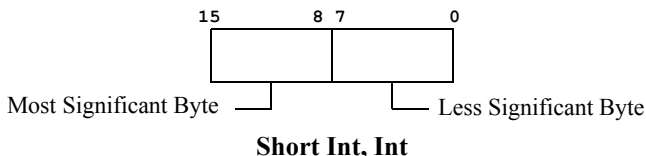
```
+pri                # enter a private region
+def _malloc=__sbreak # defines malloc as _sbreak
+new                # close region and forget malloc
libi.sm8            # load library containing _sbreak
```

This sequence has to be placed just before loading libraries, or before placing the module containing the `_sbreak` function. The private region is used to forget the `_malloc` reference once it has been aliased to `_sbreak`.

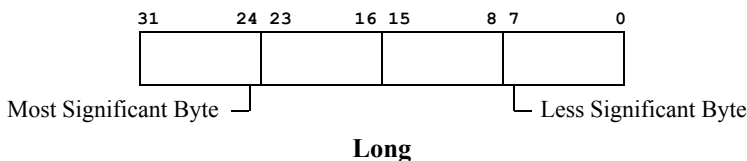
## Data Representation

Data objects of type *char* are stored as one byte. A plain *char* is defaulted to type *unsigned char*.

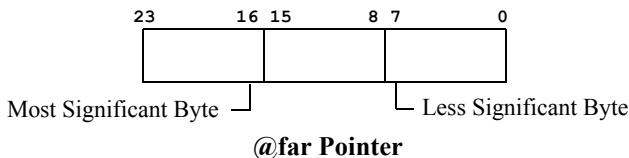
Data objects of type *short int* and *int* are stored as two bytes, more significant byte first.



Data objects of type *long int* are stored as four bytes, in descending order of significance.

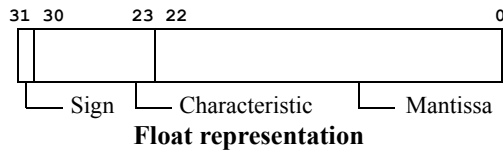


*@tiny* pointers (short range) are stored as one byte. *@near* pointers (long range) are stored as two bytes. *@far* pointers are stored as three bytes, in descending order of significance.



Data objects of type *float* are represented as for the proposed IEEE Floating Point Standard; four bytes stored in descending order of significance. The IEEE representation is: most significant bit is one for negative numbers, and zero otherwise; the next eight bits are the characteristic, biased such that the binary exponent of the number is the characteristic minus 126; the remaining bits are the fraction, starting with the weighted bit. If the characteristic is zero, the entire number is

taken as zero, and should be all zeros to avoid confusing some routines that do not process the entire number. Otherwise there is an assumed 0.5 (assertion of the weighted bit) added to all fractions to put them in the interval  $[0.5, 1.0)$ . The value of the number is the fraction, multiplied by -1 if the sign bit is set, multiplied by 2 raised to the exponent.





# Using The Compiler

---

This chapter explains how to use the C cross compiler to compile programs on your host system. It explains how to invoke the compiler, and describes its options. It also describes the functions which constitute the C library. This chapter includes the following sections:

- Invoking the Compiler
- File Naming Conventions
- Generating Listings
- Generating an Error File
- C Library Support
- Descriptions of C Library Functions

## Invoking the Compiler

To invoke the cross compiler, type the command **cxstm8**, followed by the compiler options and the name(s) of the file(s) you want to compile. All the valid compiler options are described in this chapter. Commands to compile source files have the form:

```
cxstm8 [options] <files>.[c|s]
```

**cxstm8** is the name of the *compiler*. The option list is optional. You must include the name of at least one input file *<file>*. *<file>* can be a C source file with the suffix **.c**, or an assembly language source file with the suffix **.s**. You may specify multiple input files with any combination of these suffixes in any order.

If you do not specify any command line options, **cxstm8** will compile your *<files>* with the default options. It will also write the name of each file as it is processed. It writes any error messages to STDERR.

The following command line:

```
cxstm8 +mods acia.c
```

compiles and assembles the *acia.c* C program, using the *Stack Short* model, creating the relocatable program **acia.o**.

If the compiler finds an error in your program, it halts compilation. When an error occurs, the compiler sends an error message to your terminal screen unless the option **-e** has been specified on the command line. In this case, all error messages are written to a file whose name is obtained by replacing the suffix **.c** of the source file by the suffix **.err**. An error message is still output on the terminal screen to indicate that errors have been found. **Appendix A**, “[Compiler Error Messages](#)” lists the error messages the compiler generates. If one or more command line arguments are invalid, **cxstm8** processes the next file name on the command line and begins the compilation process again.

The example command above does not specify any compiler options. In this case, the compiler will use only default options to compile and



assemble your program. You can change the operation of the compiler by specifying the options you want when you run the compiler.

To specify options to the compiler, type the appropriate option or options on the command line as shown in the first example above. Options should be separated with spaces. You must include the '-' or '+' that is part of the option name.

## Compiler Command Line Options

The **cxstm8** compiler accepts the following command line options, each of which is described in detail below:

```
cxstm8 [options] <files>
-a*> assembler options
-ce* path for errors
-cl* path for listings
-co* path for objects
-d*> define symbol
-e create error file
-ec all C files
-eg eclipse error messages
-es all assembler files
-ex* prefix executables
-f* configuration file
-g*> code generator options
-i*> path for include
-l create listing
-no do not use optimizer
-o*> optimizer options
-p*> parser options
-s create only assembler file
-sm create only dependencies
-sp create only preprocessor file
-t* path for temporary files
-v verbose
-x do not execute
+*> select compiler options
```

**Cxstm8 Option Usage**

Option	Description
<b>-a&gt;</b>	specify assembler options. Up to 128 options can be specified on the same command line. See <b>Chapter 5</b> , “ <a href="#">Using The Assembler</a> ”, for the list of all accepted options.
<b>-ce*</b>	specify a path for the error files. By default, errors are created in the same directory than the source files.
<b>-cl*</b>	specify a path for the listing files. By default, listings are created in the same directory than the source files.
<b>-co*</b>	specify a path for the object files. By default, objects are created in the same directory than the source files.
<b>-d*&gt;</b>	specify * as the name of a user-defined preprocessor symbol ( <b>#define</b> ). The form of the definition is <b>-dsymbol[=value]</b> ; the symbol is set to <b>1</b> if value is omitted. You can specify up to 128 such definitions.
<b>-e</b>	log errors from parser in a file instead of displaying them on the terminal screen. The error file name is defaulted to <code>&lt;file&gt;.err</code> , and is created only if there are errors.
<b>-ec</b>	treat all files as C source files.
<b>-eg</b>	produce error messages directly compatible with the Eclipse environment.
<b>-es</b>	treat all files as assembler source files.
<b>-ex</b>	use the compiler driver’s path as prefix to quickly locate the executable passes. Default is to use the path variable environment. This method is faster than the default behavior but reduces the command line length.
<b>-f*</b>	specify * as the name of a configuration file. This file contains a list of options, which will be automatically used by the compiler. If no file name is specified, then the compiler looks for a default configuration file named <code>cxstm8.cxf</code> in the compiler directory as specified in the installation process. For more information, see <b>Appendix B</b> , “ <a href="#">The Configuration File</a> ”.

## Cxstm8 Option Usage (cont.)

Option	Description
<b>-g*&gt;</b>	specify code generation options. Up to 128 options can be specified. See “ <a href="#">The cgstm8 Code Generator</a> ” in <b>Appendix D</b> , for the list of all accepted options.
<b>-i*&gt;</b>	define include path. You can define up to 128 different paths. Each path is a directory name, <b>not</b> terminated by any directory separator character, or a file containing an unlimited list of directory names.
<b>-l</b>	merge C source listing with assembly language code; listing output defaults to <file>.ls.
<b>-no</b>	do not use the optimizer.
<b>-o*&gt;</b>	specify optimizer options. Up to 128 options can be specified. See “ <a href="#">The costm8 Assembly Language Optimizer</a> ” in <b>Appendix D</b> , for the list of all accepted options.
<b>-p*&gt;</b>	specify parser options. Up to 128 options can be specified. See “ <a href="#">The cpstm8 Parser</a> ” in <b>Appendix D</b> , for the list of all accepted options.
<b>-s</b>	create only assembler files and stop. Do not assemble the files produced.
<b>-sm</b>	create only a list of ‘make’ compatible dependencies consisting for each source file in the object name followed by a list of header files needed to compile that file.
<b>-sp</b>	create only preprocessed files and stop. Do not compile files produced. Preprocessed output defaults to <file>.p. The produced files can be compiled as C source files.
<b>-t*</b>	specify path for temporary files. The path is a directory name, <b>not</b> terminated by any directory separator character.
<b>-v</b>	be “verbose”. Before executing a command, print the command, along with its arguments, to STDOUT. The default is to output only the names of each file processed. Each name is followed by a colon and newline.
<b>-x</b>	do not execute the passes, instead write to STDOUT the commands which otherwise would have been performed.

**Cxstm8 Option Usage (cont.)**

Option	Description
<b>+*&gt;</b>	select a predefined compiler option. These options are pre-defined in the configuration file. You can specify up to 128 compiler options on the command line. The following documents the available options as provided by the default configuration file
<b>+compact</b>	produce a smaller code but slower than the default behaviour. Smaller code is produced by enabling the optimizer factorization feature with a default depth of seven instructions.
<b>+debug</b>	produce debug information to be used by the debug utilities provided with the compiler and by any external debugger.
<b>+fast</b>	produce faster code by inlining machine library calls for long integers handling and integer switches. The code produced will be larger than without this option. For more information, see “ <a href="#">Inline Function</a> ” in <b>Chapter 3</b> .
<b>+mods</b>	select the <b>Stack Short</b> model. Variables are in <i>short range</i> memory but pointers are pointing to <i>long range</i> memory. See “ <a href="#">Memory Models for code larger than 64K</a> ” in <b>Chapter 3</b> .
<b>+modsl</b>	select the <b>Stack Long</b> mode. Variables and pointers are in and pointing to <i>long range</i> memory. See “ <a href="#">Memory Models for code larger than 64K</a> ” in <b>Chapter 3</b> .
<b>+mods0</b>	select the <b>Stack Short</b> model for application smaller than 64K. Variables are in <i>short range</i> memory but pointers are pointing to <i>long range</i> memory. See “ <a href="#">Memory Models for code smaller than 64K</a> ” in <b>Chapter 3</b> .
<b>+modsl0</b>	select the <b>Stack Long</b> model for application smaller than 64K. Variables and pointers are in and pointing to <i>long range</i> memory. See “ <a href="#">Memory Models for code smaller than 64K</a> ” in <b>Chapter 3</b> .
<b>+nobss</b>	do not use the <b>.bss</b> section for variables allocated in external memory. By default, such uninitialized variables are defined into the <i>.bss</i> section. This option is useful to force all variables to be grouped into a single section.

### Cxstm8 Option Usage (cont.)

Option	Description
<b>+nocst</b>	output literals and constants in the code section <code>.text</code> instead of the specific section <code>.const</code> .
<b>+proto</b>	enforce prototype declaration for functions. An error message is issued if a function is used and no prototype declaration is found for it. By default, the compiler accepts both syntaxes without any error.
<b>+rev</b>	reverse the bitfield filling order. By default, bitfields are filled from the Less Significant Bit (LSB) towards the Most Significant Bit (MSB) of a memory cell. If the <b>+rev</b> option is specified, bitfields are filled from the <i>msb</i> to the <i>lsb</i> .
<b>+split</b>	create a separate sub-section per function, up to a maximum number of 256 sections, thus allowing the linker to suppress unused functions if the <b>-k</b> option has been specified on at least one segment in the linker command file. For objects with more than 256 functions, the functions will be grouped together to a minimum number of functions per sub-section to not exceed the maximum of 256 sub-sections. See “ <a href="#">Segment Control Options</a> ” in <b>Chapter 6</b> .
<b>+strict</b>	enforce a stronger type checking. For more information, see “ <a href="#">Extra verifications</a> ” in <b>Appendix D</b> .
<b>+warn</b>	enable warnings. For more information, see “ <a href="#">Warning Levels</a> ” in <b>Appendix D</b> .

Note that some compiler options are specified to a default value in the configuration file, `cxstm8.cxf`, and cannot be modified without modifying the configuration file itself (see **Appendix B**, “[Modifying Compiler Operation](#)”). These options are **-pu** (char is **unsigned** by default) and **-ppb** (pack local bit variables). Take special care if you remove the **-pu** option: as libraries expect unsigned chars as argument, it is now the user responsibility to manually declare the relevant variables as **unsigned char**. If you forget this, run time errors may occur.

## File Naming Conventions

The programs making up the C cross compiler generate the following output file names, by default. See the documentation on a specific program for information about how to change the default file names accepted as input or generated as output.

Program	Input File Name	Output File Name
<b>cpstm8</b>	<file>.c	<file>.1
<b>cgstm8</b>	<file>.1	<file>.2
<b>costm8</b>	<file>.2	<file>.s
error listing	<file>.c	<file>.err
assembler listing	<file>.[c s]	<file>.ls
C header files	<file>.h	

<b>castm8</b>	<file>.s	<file>.o
source listing	<file>.s	<file>.ls

<b>clnk</b>	<file>.o	name required
-------------	----------	---------------

<b>chex</b>	<file>	STDOUT
<b>clabs</b>	<file.sm8>	<files>.la
<b>clib</b>	<file>	name required
<b>cobj</b>	<file>	STDOUT
<b>cvdwarf</b>	<file.sm8>	<file>.elf

## Generating Listings

You can generate listings of the output of any (or all) the compiler passes by specifying the **-l** option to **cxstm8**. You can locate the listing file in a different directory by using the **-cl** option.

The example program provided in the package shows the listing produced by compiling the C source file *acia.c* with the **-l** option:

```
cxstm8 +mods -l acia.c
```

## Generating an Error File

You can generate a file containing all the error messages output by the parser by specifying the **-e** option to **cxstm8**. You can locate the error file in a different directory by using the **-ce** option. For example, you would type:

```
cxstm8 +mods -e prog.c
```

The error file name is obtained from the source filename by replacing the *.c* suffix by the *.err* suffix.

## Return Status

**cxstm8** returns success if it can process all files successfully. It prints a message to STDERR and returns failure if there are errors in at least one processed file.

## Examples

To echo the names of each program that the compiler runs:

```
cxstm8 +mods -v file.c
```

To save the intermediate files created by the code generator and halt before the assembler:

```
cxstm8 +mods -s file.c
```

## C Library Support

This section describes the facilities provided by the C library. The C cross compiler for STM8 includes all useful functions for programmers writing applications for ROM-based systems.

### How C Library Functions are Packaged

The functions in the C library are packaged in three separate sub-libraries; one for machine-dependent routines (the *machine* library), one that does not support floating point (the *integer* library) and one that provides full floating point support (the *floating point* library). If your application does not perform floating point calculations, you can decrease its size and increase its runtime efficiency by including only the integer library.

### Inserting Assembler Code Directly

Assembler instructions can be quoted directly into C source files, and entered unchanged into the output assembly stream, by use of the `_asm()` function. This function is not part of any library as it is recognized by the compiler itself. See “[Inserting Inline Assembly Instructions](#)” in **Chapter 3**.

### Linking Libraries with Your Program

If your application requires floating point support, you must specify the floating point library **before** the integer library in the linker command file. Modules common to both libraries will therefore be loaded from the floating point library, followed by the appropriate modules from the floating point and integer libraries, in that order.

#### NOTE

*When using a model for application smaller than 64K, you must link with the specific set of libraries (names ending with '0').*

### Integer Library Functions

The following table lists the C library functions in the integer library.

<code>_asm</code>	<code>irq</code>	<code>malloc</code>	<code>strcat</code>
<code>abs</code>	<code>isalnum</code>	<code>memchr</code>	<code>strchr</code>
<code>atoi</code>	<code>isalpha</code>	<code>memcmp</code>	<code>strcmp</code>
<code>atol</code>	<code>isctrl</code>	<code>memcpy</code>	<code>strcpy</code>



calloc	isdigit	memmove	strcspn
checksum	isgraph	memset	strlen
checksum16	islower	printf	strncat
checksum16x	isprint	putchar	strncmp
checksumx	ispunct	puts	strncpy
div	isqrt	rand	strpbrk
eepcpy	isspace	realloc	strchr
eepset	isupper	sbreak	strspn
free	isxdigit	scanf	strstr
getchar	labs	sprintf	strtol
gets	ldiv	srand	tolower
imask	lsqrt	sscanf	toupper

## Floating Point Library Functions

The following table lists the C library functions in the float library.

acos	cosh	log	sprintf
asin	exp	log10	sqrt
atan	fabs	modf	strtod
atan2	floor	pow	tan
atof	fmod	printf	tanh
ceil	frexp	sin	
cos	ldexp	sinh	

## Common Input/Output Functions

Two of the functions that perform stream output are included in both the integer and floating point libraries. The functionalities of the versions in the integer library are a subset of the functionalities of their floating point counterparts. The versions in the integer library cannot print or manipulate floating point numbers. These functions are: *printf*, *sprintf*.

## Functions Implemented as Macros

Two of the functions in the C library are actually implemented as “macros”. Unlike other functions, which (if they do not return *int*) are declared in header files and defined in a separate object module that is linked in with your program later, functions implemented as macros are defined using **#define** preprocessor directives in the header file that declares them. Macros can therefore be used independently of any library by including the header file that defines and declares them with your program, as explained below. The functions in the C library that are implemented as macros are: *max* and *min*.

## Including Header Files

If your application calls a C library function, you must include the header file that declares the function at compile time, in order to use the proper return type and the proper function prototyping, so that all the expected arguments are properly evaluated. You do this by writing a preprocessor directive of the form:

```
#include <header_name>
```

in your program, where *<header\_name>* is the name of the appropriate header file enclosed in angle brackets. The required header file should be included before you refer to any function that it declares.

The names of the header files packaged with the C library and the functions declared in each header are listed below.

**<assert.h>** - Header file for the assertion macro: *assert*.

**<ctype.h>** - Header file for the character functions: *isalnum*, *isalpha*, *isctrl*, *isgraph*, *isprint*, *ispunct*, *isspace*, *isxdigit*, *isdigit*, *isupper*, *islower*, *tolower* and *toupper*.

**<float.h>** - Header file for limit constants for floating point values.

**<io\*.h>** - Header files for input-output registers. Each register has an upper-case name which matches the data sheet definition. The compiler provides a large set of header files for most derivative processors.

**<limits.h>** - Header file for limit constants of the compiler.

**<math.h>** - Header file for mathematical functions: *acos*, *asin*, *atan*, *atan2*, *ceil*, *cos*, *cosh*, *exp*, *fabs*, *floor*, *fmod*, *frexp*, *ldexp*, *log*, *log10*, *modf*, *pow*, *sin*, *sinh*, *sqrt*, *tan* and *tanh*.

**<processor.h>** - Header file for **inline** functions: *carry*, *irq*, *imask*.

**<stdbool.h>** - Header file for type *bool* and values *true*, *false*.

**<stddef.h>** - Header file for types: *size\_t*, *wchar\_t* and *ptrdiff\_t*.

<**stdio.h**> - Header file for stream input/output: *getchar*, *gets*, *printf*, *putchar*, *puts* and *sprintf*.

<**stdlib.h**> - Header file for general utilities: *abs*, *abort*, *atof*, *atoi*, *atol*, *div*, *exit*, *labs*, *ldiv*, *rand*, *srand*, *strtod*, *strtol* and *strtoul*.

<**string.h**> - Header file for string functions: *memchr*, *memcmp*, *memcpy*, *memmove*, *memset*, *strcat*, *strchr*, *strcmp*, *strcpy*, *strcspn*, *strlen*, *strncat*, *strncmp*, *strncpy*, *strpbrk*, *strrchr*, *strspn* and *strstr*.

**Functions returning int** - C library functions that return *int* and can therefore be called without any header file are: *isalnum*, *isalpha*, *iscntrl*, *isgraph*, *isprint*, *ispunct*, *isspace*, *isxdigit*, *isdigit*, *isupper*, *islower*, *sbreak*, *tolower* and *toupper*.

## Descriptions of C Library Functions

The following pages describe each of the functions in the C library in quick reference format. The descriptions are in alphabetical order by function name.

The *syntax* field describes the function prototype with the return type and the expected arguments, and if any, the header file name where this function has been declared.

# **`_asm`**

## **Description**

Generate inline assembly code

## **Syntax**

```
_asm("string constant", arguments...)
```

## **Function**

`_asm()` generates inline assembly code by copying *<string constant>* and quoting it into the output assembly code stream. *<arguments>* are first evaluated following the usual rules for passing arguments. The first argument is kept in the **a** register or the **x:a** register pair whenever possible, and all other arguments are pushed onto the stack. After the *<string constant>* code is output, arguments pushed to the stack are removed before to continue.

## **Return Value**

Nothing, unless `_asm()` is used in an expression. In that case, standard return conventions must be followed. See “[Register Usage](#)” in **Chapter 3**.

## **Example**

The sequence `inc x; call _main`, may be generated by the following call:

```
_asm("inc x\n call _main");
```

Note that the string-quoting syntax matches the familiar `printf()` function.

## **Notes**

`_asm()` is not packaged in any library. It is recognized by the compiler itself.

For more information, see “[Inserting Inline Assembly Instructions](#)” in **Chapter 3**.

# abort

## Description

Abort program execution

## Syntax

```
#include <stdlib.h>
void abort(void)
```

## Function

**abort** stops the program execution by calling the *exit* function which is placed by the startup module just after the call to the main function.

## Return Value

*abort* never returns.

## Example

To abort in case of error:

```
if (fatal_error)
    abort();
```

## See Also

*exit*

## Notes

*abort* is a macro equivalent to the function name *exit*.

# abs

## Description

Find absolute value

## Syntax

```
#include <stdlib.h>
int abs(int i)
```

## Function

**abs** obtains the absolute value of **i**. No check is made to see that the result can be properly represented.

## Return Value

*abs* returns the absolute value of **i**, expressed as an int.

## Example

To print out a debit or credit balance:

```
printf("balance %d%s\n", abs(bal), (bal < 0)? "CR" : "");
```

## See Also

*labs*, *fabs*

## Notes

*abs* is packaged in the integer library, and may be implemented as a **macro**.

# acos

## Description

Arccosine

## Syntax

```
#include <math.h>
double acos(double x)
```

## Function

**acos** computes the angle in radians the cosine of which is **x**, to full double precision.

## Return Value

*acos* returns the closest internal representation to *acos(x)*, expressed as a double floating value in the range [0, pi]. If **x** is outside the range [-1, 1], *acos* returns zero.

## Example

To find the arccosine of **x**:

```
theta = acos(x);
```

## See Also

*asin*, *atan*, *atan2*

## Notes

*acos* is packaged in the floating point library.

# asin

## Description

Arcsine

## Syntax

```
#include <math.h>
double asin(double x)
```

## Function

**asin** computes the angle in radians the sine of which is **x**, to full double precision.

## Return Value

*asin* returns the nearest internal representation to *asin(x)*, expressed as a double floating value in the range  $[-\pi/2, \pi/2]$ . If **x** is outside the range  $[-1, 1]$ , *asin* returns zero.

## Example

To compute the arcsine of **y**:

```
theta = asin(y);
```

## See Also

*acos*, *atan*, *atan2*

## Notes

*asin* is packaged in the floating point library.



# atan

## Description

Arctangent

## Syntax

```
#include <math.h>
double atan(double x)
```

## Function

**atan** computes the angle in radians; the tangent of which is **x**, *atan* computes the angle in radians; the tangent of which is **x**, to full double precision.

## Return Value

*atan* returns the nearest internal representation to *atan(x)*, expressed as a double floating value in the range  $[-\pi/2, \pi/2]$ .

## Example

To find the phase angle of a vector in degrees:

```
theta = atan(y/x) * 180.0 / pi;
```

## See Also

*acos*, *asin*, *atan2*

## Notes

*atan* is packaged in the floating point library.

# atan2

## Description

Arctangent of  $y/x$

## Syntax

```
#include <math.h>
double atan2(double y, double x)
```

## Function

**atan2** computes the angle in radians the tangent of which is  $y/x$  to full double precision. If  $y$  is negative, the result is negative. If  $x$  is negative, the magnitude of the result is greater than  $\pi/2$ .

## Return Value

*atan2* returns the closest internal representation to  $\text{atan}(y/x)$ , expressed as a double floating value in the range  $[-\pi, \pi]$ . If both input arguments are zero, *atan2* returns zero.

## Example

To find the phase angle of a vector in degrees:

```
theta = atan2(y/x) * 180.0/pi;
```

## See Also

*acos*, *asin*, *atan*

## Notes

*atan2* is packaged in the floating point library.

# atof

## Description

Convert buffer to double

## Syntax

```
#include <stdlib.h>
double atof(char *nptr)
```

## Function

**atof** converts the string at *nptr* into a double. The string is taken as the text representation of a decimal number, with an optional fraction and exponent. Leading whitespace is skipped and an optional sign is permitted; conversion stops on the first unrecognizable character. Acceptable inputs match the pattern:

$$[+|-]\mathbf{d}^*[\mathbf{d}^*][\mathbf{e}[+|-]\mathbf{dd}^*]$$

where **d** is any decimal digit and **e** is the character ‘e’ or ‘E’. No checks are made against overflow, underflow, or invalid character strings.

## Return Value

*atof* returns the converted double value. If the string has no recognizable characters, it returns zero.

## Example

To read a string from STDIN and convert it to a double at **d**:

```
gets(buf);
d = atof(buf);
```

## See Also

*atoi, atol, strtol, strtod*

## Notes

*atof* is packaged in the floating point library.

# atoi

## Description

Convert buffer to integer

## Syntax

```
#include <stdlib.h>
int atoi(char *nptr)
```

## Function

**atoi** converts the string at *nptr* into an integer. The string is taken as the text representation of a decimal number. Leading whitespace is skipped and an optional sign is permitted; conversion stops on the first unrecognizable character. Acceptable characters are the decimal digits. If the stop character is **I** or **L**, it is skipped over.

No checks are made against overflow or invalid character strings.

## Return Value

*atoi* returns the converted integer value. If the string has no recognizable characters, zero is returned.

## Example

To read a string from STDIN and convert it to an *int* at *i*:

```
gets(buf);
i = atoi(buf);
```

## See Also

*atof*, *atol*, *strtol*, *strtod*

## Notes

*atoi* is packaged in the integer library.

# atol

## Description

Convert buffer to long

## Syntax

```
#include <stdlib.h>
long atol(char *nptr)
```

## Function

**atol** converts the string at *nptr* into a long integer. The string is taken as the text representation of a decimal number. Leading whitespace is skipped and an optional sign is permitted; conversion stops on the first unrecognizable character. Acceptable characters are the decimal digits. If the stop character is **I** or **L** it is skipped over.

No checks are made against overflow or invalid character strings.

## Return Value

*atol* returns the converted long integer. If the string has no recognizable characters, zero is returned.

## Example

To read a string from STDIN and convert it to a long **l**:

```
gets(buf);
l = atol(buf);
```

## See Also

*atof, atoi, strtol, strtod*

## Notes

*atol* is packaged in the integer library.

# carry

## Description

Test or get the carry bit

## Syntax

```
#include <processor.h>
@inline char carry(void)
```

## Function

**carry** is an *inline* function allowing to test or get the value of the *carry* bit. When used in an *if* construct, this function expands directly to a **bcc** or **bcs** instruction. When used in an expression, it expands in order to build in the **a** register the value **0** or **1** depending on the *carry* bit value.

## Return Value

*carry* returns **0** or **1** in the **a** register if such a value is needed.

## Example

low <= 1;	produces	sll   _low
if (carry())		jruge L1
++high;		inc   _high
	L1:	
low <= 1;	produces	sll   _low
high = carry()		clr   a
		rlc   a
		ld   _high, a

## Notes

*carry* is an *inline* function and then is not defined in any library. It is therefore not possible to take its address. For more information, see “[Inline Function](#)” in **Chapter 3**.

# ceil

## Description

Round to next higher integer

## Syntax

```
#include <math.h>
double ceil(double x)
```

## Function

**ceil** computes the smallest integer greater than or equal to **x**.

## Return Value

*ceil* returns the smallest integer greater than or equal to **x**, expressed as a double floating value.

## Example

x	ceil(x)
5.1	6.0
5.0	5.0
0.0	0.0
-5.0	-5.0
-5.1	-5.0

## See Also

*floor*

## Notes

*ceil* is packaged in the floating point library.

# **\_checksum**

## **Description**

Verify the recorded checksum

## **Syntax**

```
int _checksum()
```

## **Function**

**\_checksum** scans the descriptor built by the linker and controls that the computed checksum is equal to the one expected. For more information, see “[Checksum Computation](#)” in **Chapter 6**.

## **Return Value**

*\_checksum* returns 0 if the checksum is correct, or a value different of 0 otherwise.

## **Example**

```
if (_checksum())  
    abort();
```

## **Notes**

The descriptor is built by the linker only if the *\_checksum* function is called by the application, even if there are segments marked with the **-ck** option.

*\_checksum* is packaged in the integer library.

## **See Also**

*\_checksumx*, *\_checksum16*, *\_checksum16x*



# `_checksumx`

## Description

Verify the recorded checksum

## Syntax

```
int _checksumx()
```

## Function

`_checksumx` scans the descriptor built by the linker and controls at the end that the computed 8 bit checksum is equal to the one expected. For more information, see “[Checksum Computation](#)” in **Chapter 6**.

## Return Value

`_checksumx` returns 0 if the checksum is correct, or a value different of 0 otherwise.

## Example

```
if (_checksumx())  
    abort();
```

## Notes

The descriptor is built by the linker only if the `_checksumx` function is called by the application, even if there are segments marked with the **-ck** option.

`_checksumx` is packaged in the integer library.

## See Also

`_checksum`, `_checksum16`, `_checksum16x`

# `_checksum16`

## Description

Verify the recorded checksum

## Syntax

```
int _checksum16()
```

## Function

`_checksum16` scans the descriptor built by the linker and controls at the end that the computed 16 bit checksum is equal to the one expected. For more information, see “[Checksum Computation](#)” in **Chapter 6**.

## Return Value

`_checksum16` returns 0 if the checksum is correct, or a value different of 0 otherwise.

## Example

```
if (_checksum16())  
    abort();
```

## Notes

The descriptor is built by the linker only if the `_checksum16` function is called by the application, even if there are segments marked with the **-ck** option.

`_checksum16` is packaged in the integer library.

## See Also

`_checksum`, `_checksumx`, `_checksum16x`

# `_checksum16x`

## Description

Verify the recorded checksum

## Syntax

```
int _checksum16x()
```

## Function

`_checksum16x` scans the descriptor built by the linker and controls at the end that the computed 16 bit checksum is equal to the one expected. For more information, see “[Checksum Computation](#)” in **Chapter 6**.

## Return Value

`_checksum16x` returns 0 if the checksum is correct, or a value different of 0 otherwise.

## Example

```
if (_checksum16x())  
    abort();
```

## Notes

The descriptor is built by the linker only if the `_checksum16x` function is called by the application, even if there are segments marked with the **-ck** option.

`_checksum16x` is packaged in the integer library.

## See Also

`_checksum`, `_checksumx`, `_checksum16`

# COS

## Description

Cosine

## Syntax

```
#include <math.h>
double cos(double x)
```

## Function

**cos** computes the cosine of **x**, expressed in radians, to full double precision. If the magnitude of **x** is too large to contain a fractional quadrant part, the value of *cos* is 1.

## Return Value

*cos* returns the nearest internal representation to *cos(x)* in the range [0, pi], expressed as a double floating value. A large argument may return a meaningless value.

## Example

To rotate a vector through the angle **theta**:

```
xnew = xold * cos(theta) - yold * sin(theta);
ynew = xold * sin(theta) + yold * cos(theta);
```

## See Also

*sin*, *tan*

## Notes

*cos* is packaged in the floating point library.

# cosh

## Description

Hyperbolic cosine

## Syntax

```
#include <math.h>
double cosh(double x)
```

## Function

**cosh** computes the hyperbolic cosine of **x** to full double precision.

## Return Value

*cosh* returns the nearest internal representation to *cosh(x)* expressed as a double floating value. If the result is too large to be properly represented, *cosh* returns zero.

## Example

To use the Moivre's theorem to compute  $(\cosh x + \sinh x)$  to the *n*th power:

```
demoivre = cosh(n * x) + sinh(n * x);
```

## See Also

*exp*, *sinh*, *tanh*

## Notes

*cosh* is packaged in the floating point library.

# div

## Description

Divide with quotient and remainder

## Syntax

```
#include <stdlib.h>

```

## Function

**div** divides the integer *numer* by the integer *denom* and returns the quotient and the remainder in a structure of type *div\_t*. The field *quot* contains the quotient and the field *rem* contains the remainder.

## Return Value

*div* returns a structure of type *div\_t* containing both quotient and remainder.

## Example

To get minutes and seconds from a delay in seconds:

```
div_t result;
result = div(time, 60);
min = result.quot;
sec = result.rem;
```

## See Also

*ldiv*

## Notes

*div* is packaged in the integer library.

# eeepcpy

## Description

Copy a buffer to an **eeeprom** buffer

## Syntax

```
#include <string.h>
void *eeepcpy(void *s1, void *s2, unsigned int n)
```

## Function

**eeepcpy** copies the first *n* characters starting at location *s2* into the *eeeprom* buffer beginning at *s1*.

## Return Value

Nothing.

## Example

To place “*first string, second string*” in *eeepbuf*[]):

```
eeepcpy(eeepbuf, "first string", 12);
eeepcpy(eeepbuf + 13, ", second string", 15);
```

## See Also

*eeepset*, *eeepera*

## Notes

*eeepcpy* is packaged in the integer library.

# eepera

## Description

Erase the *eprom* space

## Syntax

```
void eepera(void *s, unsigned int n)
```

## Function

**eepera** erases the first *n* bytes starting at location *s*.

## Return Value

Nothing.

## Example

To erase the *eprom* space:

```
eepera((void *)0x4000, 1024);
```

## See Also

*eepcpy*, *eeplib*

## Notes

*eepera* is a macro declared in the `<string.h>` header file, calling the *eeplib* function with argument 0.



# eepset

## Description

Propagate fill character throughout *eprom* buffer

## Syntax

```
#include <string.h>
void eepset(void *s, char c, unsigned int n)
```

## Function

**eepset** floods the *n* character buffer starting at *eprom* location *s* with fill character *c*. The function waits for all bytes to be programmed.

## Return Value

Nothing.

## Example

To flood a 512 byte *eprom* buffer with NULs:

```
eepset(eepbuf, '\0', BUFSIZ);
```

## See Also

*eencpy*, *eepera*

## Notes

*eepset* is packaged in the integer library.

# exit

## Description

Exit program execution

## Syntax

```
#include <stdlib.h>
void exit(int status)
```

## Function

**exit** stops the execution of a program by switching to the startup module just after the call to the *main* function. The status argument is not used by the current implementation.

## Return Value

*exit* never returns.

## Example

To *exit* in case of error:

```
if (fatal_error)
    exit();
```

## See Also

*abort*

## Notes

*exit* is in the startup module.

# exp

## Description

Exponential

## Syntax

```
#include <math.h>
double exp(double x)
```

## Function

**exp** computes the exponential of **x** to full double precision.

## Return Value

*exp* returns the nearest internal representation to *exp x*, expressed as a double floating value. If the result is too large to be properly represented, *exp* returns zero.

## Example

To compute the hyperbolic sine of **x**:

```
sinh = (exp(x) - exp(-x)) / 2.0;
```

## See Also

*log*

## Notes

*exp* is packaged in the floating point library.

# fabs

## Description

Find double absolute value

## Syntax

```
#include <math.h>
double fabs(double x)
```

## Function

**fabs** obtains the absolute value of **x**.

## Return Value

*fabs* returns the absolute value of **x**, expressed as a double floating value.

## Example

x	fabs(x)
5.0	5.0
0.0	0.0
-3.7	3.7

## See Also

*abs*, *labs*

## Notes

*fabs* is packaged in the floating point library.

# `_fctcpy`

## Description

Copy a moveable code segment in RAM

## Syntax

```
int _fctcpy(char name);
```

## Function

`_fctcpy` copies a moveable code segment in RAM from its storage location in ROM. `_fctcpy` scans the descriptor built by the linker and looks for a moveable segment whose flag byte matches the given argument. If such a segment is found, it is entirely copied in RAM. Any function defined in that segment may then be called directly. For more information, see “[Moveable Code](#)” in **Chapter 6**.

## Return Value

`_fctcpy` returns a non zero value if a segment has been found and copied. It returns 0 otherwise.

## Example

```
if (_fctcpy('b'))  
    flash();
```

## Notes

`_fctcpy` is packaged in the machine library.

# floor

## Description

Round to next lower integer

## Syntax

```
#include <math.h>
double floor(double x)
```

## Function

**floor** computes the largest integer less than or equal to **x**.

## Return Value

*floor* returns the largest integer less than or equal to **x**, expressed as a double floating value.

## Example

x	floor(x)
5.1	5.0
5.0	5.0
0.0	0.0
-5.0	-5.0
-5.1	-6.0

## See Also

*ceil*

## Notes

*floor* is packaged in the floating point library.

# fmod

## Description

Find double modulus

## Syntax

```
#include <math.h>
double fmod(double x, double y)
```

## Function

**fmod** computes the floating point remainder of **x** / **y**, to full double precision. The return value of **f** is determined using the formula:

$$\mathbf{f} = \mathbf{x} - \mathbf{i} * \mathbf{y}$$

where **i** is some integer, **f** is the same sign as **x**, and the absolute value of **f** is less than the absolute value of **y**.

## Return Value

*fmod* returns the value of **f** expressed as a double floating value. If **y** is zero, *fmod* returns zero.

## Example

x	y	fmod(x, y)
5.5	5.0	0.5
5.0	5.0	0.0
0.0	0.0	0.0
-5.5	5.0	-0.5

## Notes

*fmod* is packaged in the floating point library.

# frexp

## Description

Extract fraction from exponent part

## Syntax

```
#include <math.h>
double frexp(double val, int *exp)
```

## Function

**frexp** partitions the double at *val*, which should be non-zero, into a fraction in the interval  $[1/2, 1)$  times two raised to an integer power. It then delivers the integer power to *\*exp*, and returns the fractional portion as the value of the function. The exponent is generally meaningless if *val* is zero.

## Return Value

*frexp* returns the power of two fraction of the double at *val* as the return value of the function, and writes the exponent at *\*exp*.

## Example

To implement the *sqrt(x)* function:

```
double sqrt(double x)
{
    extern double newton(double);
    int n;

    x = frexp(x, &n);
    x = newton(x);
    if (n & 1)
        x *= SQRT2;
    return (ldexp(x, n / 2));
}
```

## See Also

*ldexp*

## Notes

*frexp* is packaged in the floating point library.



# getchar

## Description

Get character from input stream

## Syntax

```
#include <stdio.h>
int getchar(void)
```

## Function

**getchar** obtains the next input character, if any, from the user supplied input stream. This user must rewrite this function in C or in assembly language to provide an interface to the input mechanism of the C library.

## Return Value

*getchar* returns the next character from the input stream. If end of file (break) is encountered, or a read error occurs, *getchar* returns EOF.

## Example

To copy characters from the input stream to the output stream:

```
while ((c = getchar()) != EOF)
    putchar(c);
```

## See Also

*putchar*

## Notes

*getchar* is packaged in the integer library, and is by default using the first serial port **SCI 1**.

# gets

## Description

Get a text line from input stream

## Syntax

```
#include <stdio.h>
char *gets(char *s)
```

## Function

**gets** copies characters from the input stream to the buffer starting at **s**. Characters are copied until a newline is reached or end of file is reached. If a newline is reached, it is discarded and a NUL is written immediately following the last character read into **s**.

*gets* uses *getchar* to read each character.

## Return Value

*gets* returns **s** if successful. If end of file is reached, *gets* returns NULL. If a read error occurs, the array contents are indeterminate and *gets* returns NULL.

## Example

To copy input to output, line by line:

```
while (puts(gets(buf)))
    ;
```

## See Also

*puts*

## Notes

There is no assured limit on the size of the line read by *gets*.

*gets* is packaged in the integer library.

# imask

## Description

Test the interrupt mask bit

## Syntax

```
#include <processor.h>
@inline char imask(void)
```

## Function

**imask** is an *inline* function allowing to test the interrupt mask bit. The *imask* function can only be used in an *if* construct. This function expands directly to a **bms** or **bmc** instruction.

## Return Value

None.

## Example

if (imask())	produces	jrnms L3
++high;		inc _high
		L3:
if (!imask())	produces	bms L1
++high		inc _high
		L1:

## Notes

*imask* is an *inline* function and then is not defined in any library. It is therefore not possible to take its address. For more information, see “[Inline Function](#)” in **Chapter 3**.

# irq

## Description

Test the interrupt line level

## Syntax

```
#include <processor.h>
@inline char irq(void)
```

## Function

**irq** is an *inline* function allowing to test the interrupt line level. The *irq* function can only be used in an *if* construct. This function expands directly to a **bih** or **bil** instruction.

## Return Value

None.

## Example

if (irq())	produces	jril L3
++high;		inc _high
	L3:	
if (!irq())	produces	bih L1
++high		inc _high
	L1:	

## Notes

*irq* is an *inline* function and then is not defined in any library. It is therefore not possible to take its address. For more information, see “[Inline Function](#)” in **Chapter 3**.

# isalnum

## Description

Test for alphabetic or numeric character

## Syntax

```
#include <ctype.h>
int isalnum(char c)
```

## Function

**isalnum** tests whether **c** is an alphabetic character (either upper or lower case), or a decimal digit.

## Return Value

*isalnum* returns nonzero if the argument is an alphabetic or numeric character; otherwise the value returned is zero.

## Example

To test for a valid C identifier:

```
if (isalpha(*s) || *s == '_')
    for (++s; isalnum(*s) || *s == '_'; ++s)
        ;
```

## See Also

*isalpha*, *isdigit*, *islower*, *isupper*, *isxdigit*, *tolower*, *toupper*

## Notes

If the argument is outside the range [-1, 255], the result is undefined.

*isalnum* is packaged in the integer library.

# isalpha

## Description

Test for alphabetic character

## Syntax

```
#include <ctype.h>
int isalpha(char c)
```

## Function

**isalpha** tests whether **c** is an alphabetic character, either upper or lower case.

## Return Value

*isalpha* returns nonzero if the argument is an alphabetic character. Otherwise the value returned is zero.

## Example

To find the end points of an alphabetic string:

```
while (*first && !isalpha(*first))
    ++first;
for (last = first; isalpha(*last); ++last)
    ;
```

## See Also

*isalnum*, *isdigit*, *islower*, *isupper*, *isxdigit*, *tolower*, *toupper*

## Notes

If the argument is outside the range [-1, 255], the result is undefined.

*isalpha* is packaged in the integer library.

# isctrl

## Description

Test for control character

## Syntax

```
#include <ctype.h>
int isctrl(char c)
```

## Function

**isctrl** tests whether **c** is a delete character (0177 in ASCII), or an ordinary control character (less than 040 in ASCII).

## Return Value

*isctrl* returns nonzero if **c** is a control character; otherwise the value is zero.

## Example

To map control characters to percent signs:

```
for (; *s; ++s)
    if (isctrl(*s))
        *s = '%';
```

## See Also

*isgraph*, *isprint*, *ispunct*, *isspace*

## Notes

If the argument is outside the range [-1, 255], the result is undefined.

*isctrl* is packaged in the integer library.

# isdigit

## Description

Test for digit

## Syntax

```
#include <ctype.h>
int isdigit(char c)
```

## Function

**isdigit** tests whether **c** is a decimal digit.

## Return Value

*isdigit* returns nonzero if **c** is a decimal digit; otherwise the value returned is zero.

## Example

To convert a decimal digit string to a number:

```
for (sum = 0; isdigit(*s); ++s)
    sum = sum * 10 + *s - '0';
```

## See Also

*isalnum, isalpha, islower, isupper, isxdigit, tolower, toupper*

## Notes

If the argument is outside the range [-1, 255], the result is undefined.

*isdigit* is packaged in the integer library.



# isgraph

## Description

Test for graphic character

## Syntax

```
#include <ctype.h>
int isgraph(char c)
```

## Function

**isgraph** tests whether **c** is a graphic character; i.e. any printing character except a space (040 in ASCII).

## Return Value

*isgraph* returns nonzero if **c** is a graphic character. Otherwise the value returned is zero.

## Example

To output only graphic characters:

```
for (; *s; ++s)
    if (isgraph(*s))
        putchar(*s);
```

## See Also

*isctrl*, *isprint*, *ispunct*, *isspace*

## Notes

If the argument is outside the range [-1, 255], the result is undefined.

*isgraph* is packaged in the integer library.

# islower

## Description

Test for lowercase character

## Syntax

```
#include <ctype.h>
int islower(char c)
```

## Function

**islower** tests whether **c** is a lowercase alphabetic character.

## Return Value

*islower* returns nonzero if **c** is a lowercase character; otherwise the value returned is zero.

## Example

To convert to uppercase:

```
if (islower(c))
    c += 'A' - 'a';    /* also see toupper() */
```

## See Also

*isalnum*, *isalpha*, *isdigit*, *isupper*, *isxdigit*, *tolower*, *toupper*

## Notes

If the argument is outside the range [-1, 255], the result is undefined.

*islower* is packaged in the integer library.

# isprint

## Description

Test for printing character

## Syntax

```
#include <ctype.h>
int isprint(char c)
```

## Function

**isprint** tests whether **c** is any printing character. Printing characters are all characters between a space (040 in ASCII) and a tilde '~' character (0176 in ASCII).

## Return Value

*isprint* returns nonzero if **c** is a printing character; otherwise the value returned is zero.

## Example

To output only printable characters:

```
for (; *s; ++s)
    if (isprint(*s))
        putchar(*s);
```

## See Also

*isctrl*, *isgraph*, *ispunct*, *isspace*

## Notes

If the argument is outside the range [-1, 255], the result is undefined.

*isprint* is packaged in the integer library.

# ispunct

## Description

Test for punctuation character

## Syntax

```
#include <ctype.h>
int ispunct(char c)
```

## Function

**ispunct** tests whether **c** is a punctuation character. Punctuation characters include any printing character except space, a digit, or a letter.

## Return Value

*ispunct* returns nonzero if **c** is a punctuation character; otherwise the value returned is zero.

## Example

To collect all punctuation characters in a string into a buffer:

```
for (i = 0; *s; ++s)
    if (ispunct(*s))
        buf[i++] = *s;
```

## See Also

*isctrl*, *isgraph*, *isprint*, *isspace*

## Notes

If the argument is outside the range [-1, 255], the result is undefined.

*ispunct* is packaged in the integer library.

# isspace

## Description

Test for whitespace character

## Syntax

```
#include <ctype.h>
int isspace(char c)
```

## Function

**isspace** tests whether **c** is a whitespace character. Whitespace characters are horizontal tab (`'\t'`), newline (`'\n'`), vertical tab (`'\v'`), form feed (`'\f'`), carriage return (`'\r'`), and space (`' '`).

## Return Value

*isspace* returns nonzero if **c** is a whitespace character; otherwise the value returned is zero.

## Example

To skip leading whitespace:

```
while (isspace(*s))
    ++s;
```

## See Also

*isctrl*, *isgraph*, *isprint*, *ispunct*

## Notes

If the argument is outside the range  $[-1, 255]$ , the result is undefined.

*isspace* is packaged in the integer library.

# isupper

## Description

Test for uppercase character

## Syntax

```
int isupper(char c)
```

## Function

**isupper** tests whether **c** is an uppercase alphabetic character.

## Return Value

*isupper* returns nonzero if **c** is an uppercase character; otherwise the value returned is zero.

## Example

To convert to lowercase:

```
if (isupper(c))
    c += 'a' - 'A'; /* also see tolower() */
```

## See Also

*isalnum*, *isalpha*, *isdigit*, *islower*, *isxdigit*, *tolower*, *toupper*

## Notes

If the argument is outside the range [-1, 255], the result is undefined.

*isupper* is packaged in the integer library.

# isxdigit

## Description

Test for hexadecimal digit

## Syntax

```
#include <ctype.h>
int isxdigit(char c)
```

## Function

**isxdigit** tests whether **c** is a hexadecimal digit, *i.e.* in the set [0123456789abcdefABCDEF].

## Return Value

*isxdigit* returns nonzero if **c** is a hexadecimal digit; otherwise the value returned is zero.

## Example

To accumulate a hexadecimal digit:

```
for (sum = 0; isxdigit(*s); ++s)
    if (isdigit(*s))
        sum = sum * 10 + *s - '0';
    else
        sum = sum * 10 + tolower(*s) + (10 - 'a');
```

## See Also

*isalnum, isalpha, isdigit, islower, isupper, tolower, toupper*

## Notes

If the argument is outside the range [-1, 255], the result is undefined.

*isxdigit* is packaged in the integer library.

# labs

## Description

Find long absolute value

## Syntax

```
#include <stdlib.h>
long labs(long l)
```

## Function

**labs** obtains the absolute value of **l**. No check is made to see that the result can be properly represented.

## Return Value

*labs* returns the absolute value of **l**, expressed as an long int.

## Example

To print out a debit or credit balance:

```
printf("balance %ld%s\n",labs(bal),(bal < 0) ? "CR" : "");
```

## See Also

*abs*, *fabs*

## Notes

*labs* is packaged in the integer library.



# ldexp

## Description

Scale double exponent

## Syntax

```
#include <math.h>

double ldexp(double x, int exp)
```

## Function

**ldexp** multiplies the double **x** by two raised to the integer power **exp**.

## Return Value

*ldexp* returns the double result  $x * (1 \ll \text{exp})$  expressed as a double floating value. If a range error occurs, *ldexp* returns **HUGE\_VAL**.

## Example

x	exp	ldexp(x, exp)
1.0	1	2.0
1.0	0	1.0
1.0	-1	0.5
0.0	0	0.0

## See Also

*frexp*, *modf*

## Notes

*ldexp* is packaged in the floating point library.

# ldiv

## Description

Long divide with quotient and remainder

## Syntax

```
#include <stdlib.h>
ldiv_t ldiv(long numer, long denom)
```

## Function

**ldiv** divides the long integer *numer* by the long integer *denom* and returns the quotient and the remainder in a structure of type *ldiv\_t*. The field *quot* contains the quotient and the field *rem* contains the remainder.

## Return Value

*ldiv* returns a structure of type *ldiv\_t* containing both quotient and remainder.

## Example

To get minutes and seconds from a delay in seconds:

```
ldiv_t result;
result = ldiv(time, 60L);
min = result.quot;
sec = result.rem;
```

## See Also

*div*

## Notes

*ldiv* is packaged in the integer library.

# log

## Description

Natural logarithm

## Syntax

```
#include <math.h>
double log(double x)
```

## Function

**log** computes the natural logarithm of **x** to full double precision.

## Return Value

*log* returns the closest internal representation to  $\log(x)$ , expressed as a double floating value. If the input argument is less than zero, or is too large to be represented, *log* returns zero.

## Example

To compute the hyperbolic arccosine of **x**:

```
arccosh = log(x + sqrt(x * x - 1));
```

## See Also

*exp*

## Notes

*log* is packaged in the floating point library.

# log10

## Description

Common logarithm

## Syntax

```
#include <math.h>
double log10(double x)
```

## Function

**log10** computes the common log of **x** to full double precision by computing the natural log of **x** divided by the natural log of 10. If the input argument is less than zero, a domain error will occur. If the input argument is zero, a range error will occur.

## Return Value

*log10* returns the nearest internal representation to *log10* **x**, expressed as a double floating value. If the input argument is less than or equal to zero, *log10* returns zero.

## Example

To determine the number of digits in **x**, where **x** is a positive integer expressed as a double:

```
ndig = log10(x) + 1;
```

## See Also

*log*

## Notes

*log10* is packaged in the floating point library.

# max

## Description

Test for maximum

## Syntax

```
#include <stdlib.h>
max(a,b)
```

## Function

**max** obtains the maximum of its two arguments, **a** and **b**. Since *max* is implemented as a C preprocessor macro, its arguments can be any numerical type, and type coercion occurs automatically.

## Return Value

*max* is a numerical rvalue of the form  $((a < b) ? b : a)$ , suitably parenthesized.

## Example

To set a new maximum level:

```
hiwater = max(hiwater, level);
```

## See Also

*min*

## Notes

*max* is an extension to the proposed ANSI C standard.

*max* is a macro declared in the *<stdlib.h>* header file. You can use it by including *<stdlib.h>* with your program. Because it is a macro, *max* cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated other than once.

# memchr

## Description

Scan buffer for character

## Syntax

```
#include <string.h>
void *memchr(void *s, char c, unsigned char n)
```

## Function

**memchr** looks for the first occurrence of a specific character **c** in an **n** character buffer starting at **s**.

## Return Value

*memchr* returns a pointer to the first character that matches **c**, or NULL if no character matches.

## Example

To map *keybuf[]* characters into *subst[]* characters:

```
if ((t = memchr(keybuf, *s, KEYSIZ)) != NULL)
    *s = subst[t - keybuf];
```

## See Also

*strchr*, *strcspn*, *strpbrk*, *strrchr*, *strspn*

## Notes

*memchr* is packaged in the integer library.

# memcmp

## Description

Compare two buffers for lexical order

## Syntax

```
#include <string.h>
int memcmp(void *s1, void *s2, unsigned char n)
```

## Function

**memcmp** compares two text buffers, character by character, for lexical order in the character collating sequence. The first buffer starts at **s1**, the second at **s2**; both buffers are **n** characters long.

## Return Value

*memcmp* returns a short integer greater than, equal to, or less than zero, according to whether **s1** is lexicographically greater than, equal to, or less than **s2**.

## Example

To look for the string “*include*” in name:

```
if (memcmp(name, "include", 7) == 0)
    doinclude();
```

## See Also

*strcmp*, *strncmp*

## Notes

*memcmp* is packaged in the integer library.

# memcpy

## Description

Copy one buffer to another

## Syntax

```
#include <string.h>
void *memcpy(void *s1, void *s2, unsigned char n)
```

## Function

**memcpy** copies the first **n** characters starting at location **s2** into the buffer beginning at **s1**.

## Return Value

*memcpy* returns **s1**.

## Example

To place “*first string, second string*” in *buf[]*:

```
memcpy(buf, "first string", 12);
memcpy(buf + 13, ", second string", 15);
```

## See Also

*strcpy*, *strncpy*

## Notes

*memcpy* is packaged in the *integer* library and may be implemented as an *inline* function.



# memmove

## Description

Copy one buffer to another

## Syntax

```
#include <string.h>
void *memmove(void *s1, void *s2, unsigned char n)
```

## Function

**memmove** copies the first **n** characters starting at location **s2** into the buffer beginning at **s1**. If the two buffers overlap, the function performs the copy in the appropriate sequence, so the copy is not corrupted.

## Return Value

*memmove* returns **s1**.

## Example

To shift an array of characters:

```
memmove(buf, &buf[5], 10);
```

## See Also

*memcpy*

## Notes

*memmove* is packaged in the integer library.

# memset

## Description

Propagate fill character throughout buffer

## Syntax

```
#include <string.h>
void *memset(void *s, char c, unsigned char n)
```

## Function

**memset** floods the **n** character buffer starting at **s** with fill character **c**.

## Return Value

*memset* returns **s**.

## Example

To flood a 512-byte buffer with NULs:

```
memset(buf, '\0', BUFSIZ);
```

## Notes

*memset* is packaged in the integer library and may be implemented as an *inline* function.

# min

## Description

Test for minimum

## Syntax

```
#include <stdlib.h>
min(a, b)
```

## Function

**min** obtains the minimum of its two arguments, **a** and **b**. Since *min* is implemented as a C preprocessor macro, its arguments can be any numerical type, and type coercion occurs automatically.

## Return Value

*min* is a numerical rvalue of the form  $((a < b) ? a : b)$ , suitably parenthesized.

## Example

To set a new minimum level:

```
nmove = min(space, size);
```

## See Also

*max*

## Notes

*min* is an extension to the ANSI C standard.

*min* is a macro declared in the *<stdlib.h>* header file. You can use it by including *<stdlib.h>* with your program. Because it is a macro, *min* cannot be called from non-C programs, nor can its address be taken. Arguments with side effects may be evaluated more than once.

# modf

## Description

Extract fraction and integer from double

## Syntax

```
#include <math.h>
double modf(double val, double *pd)
```

## Function

**modf** partitions the double *val* into an integer portion, which is delivered to *\*pd*, and a fractional portion, which is returned as the value of the function. If the integer portion cannot be represented properly in an int, the result is truncated on the left without complaint.

## Return Value

*modf* returns the signed fractional portion of *val* as a double floating value, and writes the integer portion at *\*pd*.

## Example

val	*pd	modf(val, *pd)
5.1	5	0.1
5.0	5	0.0
4.9	4	0.9
0.0	0	0.0
-1.4	-1	-0.4

## See Also

*frexp*, *ldexp*

## Notes

*modf* is packaged in the floating point library.

# pow

## Description

Raise *x* to the *y* power

## Syntax

```
#include <math.h>
double pow(double x, double y)
```

## Function

**pow** computes the value of *x* raised to the power of *y*.

## Return Value

*pow* returns the value of *x* raised to the power of *y*, expressed as a double floating value. If *x* is zero and *y* is less than or equal to zero, or if *x* is negative and *y* is not an integer, *pow* returns zero.

## Example

x	y	pow(x, y)
2.0	2.0	4.0
2.0	1.0	2.0
2.0	0.0	1.0
1.0	any	1.0
0.0	-2.0	0
-1.0	2.0	1.0
-1.0	2.1	0

## See Also

*exp*

## Notes

*pow* is packaged in the floating point library.

# printf

## Description

Output formatted arguments to stdout

## Syntax

```
#include <stdio.h>
int printf(@near char *fmt, ...)
```

## Function

**printf** writes formatted output to the output stream using the format string at *fmt* and the arguments specified by ..., as described below.

*printf* uses *putchar* to output each character.

## Format Specifiers

The format string at *fmt* consists of literal text to be output, interspersed with conversion specifications that determine how the arguments are to be interpreted and how they are to be converted for output. If there are insufficient arguments for the format, the results are undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. *printf* returns when the end of the format string is encountered.

Each *<conversion specification>* is started by the character ‘%’. After the ‘%’, the following appear in sequence:

**<flags>** - zero or more which modify the meaning of the conversion specification.

**<field width>** - a decimal number which optionally specifies a minimum field width. If the converted value has fewer characters than the field width, it is padded on the left (or right, if the left adjustment flag has been given) to the field width. The padding is with spaces unless the field width digit string starts with zero, in which case the padding is with zeros.

**<precision>** - a decimal number which specifies the minimum number of digits to appear for **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal point for **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of characters to be printed from a string in an **s** conversion. The precision takes the form of a period followed by a decimal digit string. A null digit string is treated as zero.

**h** - optionally specifies that the following **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a short int or unsigned short int argument (the argument will have been widened according to the integral widening conversions, and its value must be cast to short or unsigned short before printing). It specifies a short pointer argument if associated with the **p** conversion character. If an **h** appears with any other conversion character, it is ignored.

**l** - optionally specifies that the **d**, **i**, **o**, **u**, **x**, and **X** conversion character applies to a long int or unsigned long int argument. It specifies a long or far pointer argument if used with the **p** conversion character. If the **l** appears with any other conversion character, it is ignored.

**L** - optionally specifies that the following **e**, **E**, **f**, **g**, and **G** conversion character applies to a long double argument. If the **L** appears with any other conversion character, it is ignored.

**<conversion character>** - character that indicates the type of conversion to be applied.

A field width or precision, or both, may be indicated by an asterisk '\*' instead of a digit string. In this case, an int argument supplies the field width or precision. The arguments supplying field width must appear before the optional argument to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if it were missing.

The **<flags>** field is zero or more of the following:

**space** - a space will be prepended if the first character of a signed conversion is not a sign. This flag will be ignored if space and + flags are both specified.

**#** - result is to be converted to an “alternate form”. For **c**, **d**, **i**, **s**, and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be zero. For **p**, **x** and **X** conversion, a non-zero result will have **Ox** or **OX** prepended to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will contain a decimal point, even if no digits follow the point. For **g** and **G** conversions, trailing zeros will not be removed from the result, as they normally are. For **p** conversion, it designates hexadecimal output.

**+** - result of signed conversion will begin with a plus or minus sign.

**-** - result of conversion will be left justified within the field.

The *<conversion character>* is one of the following:

**%** - a ‘%’ is printed. No argument is converted.

**c** - the char argument is converted to a character and printed.

**d**, **i**, **o**, **u**, **x**, **X** - the int argument is converted to signed decimal (**d** or **i**), unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or **X**); the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** are used for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is **1**. The result of converting a zero value with precision of zero is no characters.

**e**, **E** - the double argument is converted in the style **[-]d.ddde+dd**, where there is one digit before the decimal point and the number of digits after it is equal to the precision. If the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits. However, if the magnitude to be printed is greater than or equal to **1E+100**, additional exponent digits will be printed as necessary.

**f** - the double argument is converted to decimal notation in the style **[-]ddd.ddd**, where the number of digits following the decimal point is equal to the precision specification. If the precision is missing, it is



taken as 6. If the precision is explicitly zero, no decimal point appears. If a decimal point appears, at least one digit appears before it.

**g, G** - the double argument is printed in style **f** or **e** (or in style **E** in the case of a **G** format code), with the precision specifying the number of significant digits. The style used depends on the value converted; style **e** will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeros are removed from the result; a decimal point appears only if it is followed by a digit.

**n** - the argument is taken to be an `int *` pointer to an integer into which is written the number of characters written to the output stream so far by this call to *printf*. No argument is converted.

**p** - the argument is taken to be a `void *` pointer to an object. The value of the pointer is converted to a sequence of printable characters, and printed as a hexadecimal number with the number of digits printed being determined by the field width.

**s, S** - the argument is taken to be a `char *` pointer to a string. When the **Compact** model is used, the **s** format will use a **@tiny** pointer and the **S** format will use a **@near** pointer. Characters from the string are written up to, but not including, the terminating NUL, or until the number of characters indicated by the precision are written. If the precision is missing, it is taken to be arbitrarily large, so all characters before the first NUL are printed.

If the character after '%' is not a valid conversion character, the behavior is undefined.

If any argument is or points to an aggregate (except for an array of characters using **%s** conversion or any pointer using **%p** conversion), unpredictable results will occur.

A nonexistent or small field width does not cause truncation of a field; if the result is wider than the field width, the field is expanded to contain the conversion result.

## Return Value

*printf* returns the number of characters transmitted, or a negative number if a write error occurs.

## Notes

A call with more conversion specifiers than argument variables will cause unpredictable results.

## Example

To print **arg**, which is a double with the value **5100.53**:

```
printf("%8.2f\n", arg);  
printf("%*.*f\n", 8, 2, arg);
```

both forms will output: **05100.53**

## See Also

*sprintf*

## Notes

*printf* is packaged in both the integer library and the floating point library. The functionality of the integer only version of *printf* is a subset of the functionality of the floating point version. The integer only version cannot print or manipulate floating point numbers. If your programs call the integer only version of *printf*, the following conversion specifiers are invalid: **e**, **E**, **f**, **g** and **G**. The **L** modifier is also invalid.

If *printf* encounters an invalid conversion specifier, the invalid specifier is ignored and no special message is generated.

# putchar

## Description

Put a character to output stream

## Syntax

```
#include <stdio.h>
int putchar(char c)
```

## Function

**putchar** copies **c** to the user specified output stream.

You must rewrite *putchar* in either C or assembly language to provide an interface to the output mechanism to the C library.

## Return Value

*putchar* returns **c**. If a write error occurs, *putchar* returns EOF.

## Example

To copy input to output:

```
while ((c = getchar()) != EOF)
    putchar(c);
```

## See Also

*getchar*

## Notes

*putchar* is packaged in the integer library, and is by default using the first serial port **SCI 1**.

# puts

## Description

Put a text line to output stream

## Syntax

```
#include <stdio.h>
int puts(char *s)
```

## Function

**puts** copies characters from the buffer starting at **s** to the output stream and appends a newline character to the output stream.

*puts* uses *putchar* to output each character. The terminating NUL is not copied.

## Return Value

*puts* returns zero if successful, or else nonzero if a write error occurs.

## Example

To copy input to output, line by line:

```
while (puts(gets(buf)))
    ;
```

## See Also

*gets*

## Notes

*puts* is packaged in the integer library.

# rand

## Description

Generate pseudo-random number

## Syntax

```
#include <stdlib.h>
int rand(void)
```

## Function

**rand** computes successive pseudo-random integers in the range [0, 32767], using a linear multiplicative algorithm which has a period of 2 raised to the power of 32.

## Example

```
int dice()
{
    return (rand() % 6 + 1);
}
```

## Return Value

*rand* returns a pseudo-random integer.

## See Also

*srand*

## Notes

*rand* is packaged in the integer library.

# sin

## Description

Sin

## Syntax

```
#include <math.h>
double sin(double x)
```

## Function

**sin** computes the sine of **x**, expressed in radians, to full double precision. If the magnitude of **x** is too large to contain a fractional quadrant part, the value of sin is 0.

## Return Value

*sin* returns the closest internal representation to *sin(x)* in the range  $[-\pi/2, \pi/2]$ , expressed as a double floating value. A large argument may return a meaningless result.

## Example

To rotate a vector through the angle *theta*:

```
xnew = xold * cos(theta) - yold * sin(theta);
ynew = xold * sin(theta) + yold * cos(theta);
```

## See Also

*cos*, *tan*

## Notes

*sin* is packaged in the floating point library.

# sinh

## Description

Hyperbolic sine

## Syntax

```
#include <math.h>
double sinh(double x)
```

## Function

**sinh** computes the hyperbolic sine of **x** to full double precision.

## Return Value

*sinh* returns the closest internal representation to  $\sinh(x)$ , expressed as a double floating value. If the result is too large to be properly represented, *sinh* returns zero.

## Example

To obtain the hyperbolic sine of complex **z**:

```
typedef struct
{
    double x, iy;
}complex;

complex z;

z.x = sinh(z.x) * cos(z.iy);
z.iy = cosh(z.x) * sin(z.iy);
```

## See Also

*cosh*, *exp*, *tanh*

## Notes

*sinh* is packaged in the floating point library.

# sprintf

## Description

Output arguments formatted to buffer

## Syntax

```
#include <stdio.h>
int sprintf(char *s, @near char fmt, ...)
```

## Function

**sprintf** writes formatted to the buffer pointed at by **s** using the format string at *fmt* and the arguments specified by ..., in exactly the same way as *printf*. See the description of the *printf* function for information on the format conversion specifiers. A NUL character is written after the last character in the buffer.

## Return Value

*sprintf* returns the numbers of characters written, not including the terminating NUL character.

## Example

To format a double at **d** into *buf*:

```
sprintf(buf, "%10f\n", d);
```

## See Also

*printf*

## Notes

*sprintf* is packaged in both the integer library and the floating point library. The functionality of the integer only version of *sprintf* is a subset of the functionality of the floating point version. The integer only version cannot print or manipulate floating point numbers. If your programs call the integer only version of *sprintf*, the following conversion specifiers are invalid: **e**, **E**, **f**, **g** and **G**. The **L** flag is also invalid.



# sqrt

## Description

Real square root

## Syntax

```
#include <math.h>
double sqrt(double x)
```

## Function

**sqrt** computes the square root of **x** to full double precision.

## Return Value

*sqrt* returns the nearest internal representation to *sqrt(x)*, expressed as a double floating value. If **x** is negative, *sqrt* returns zero.

## Example

To use *sqrt* to check whether **n > 2** is a prime number:

```
if (!(n & 01))
    return (NOTPRIME);
sq = sqrt((double)n);
for (div = 3; div <= sq; div += 2)
    if (!(n % div))
        return (NOTPRIME);
return (PRIME);
```

## Notes

*sqrt* is packaged in the floating point library.

# srand

## Description

Seed pseudo-random number generator

## Syntax

```
#include <stdlib.h>
void srand(unsigned char nseed)
```

## Function

**srand** uses *nseed* as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to *rand*. If *srand* is called with the same seed value, the sequence of pseudo-random numbers will be repeated. The initial seed value used by *rand* and *srand* is 1.

## Return Value

Nothing.

## Example

To set up a new sequence of random numbers:

```
srand(103);
```

## See Also

*rand*

## Notes

*srand* is packaged in the integer library.

# strcat

## Description

Concatenate strings

## Syntax

```
#include <string.h>
char *strcat(char *s1, char *s2)
```

## Function

**strcat** appends a copy of the NUL terminated string at **s2** to the end of the NUL terminated string at **s1**. The first character of **s2** overlaps the NUL at the end of **s1**. A terminating NUL is always appended to **s1**.

## Return Value

*strcat* returns **s1**.

## Example

To place the strings “*first string*, *second string*” in *buf*[]):

```
buf[0] = '\0';
strcpy(buf, "first string");
strcat(buf, ", second string");
```

## See Also

*strncat*

## Notes

There is no way to specify the size of the destination area to prevent storage overwrites.

*strcat* is packaged in the integer library.

# strchr

## Description

Scan string for first occurrence of character

## Syntax

```
#include <string.h>
char *strchr(char *s1, char c)
```

## Function

**strchr** looks for the first occurrence of a specific character **c** in a NUL terminated target string **s**.

## Return Value

*strchr* returns a pointer to the first character that matches **c**, or NULL if none does.

## Example

To map *keystr*[] characters into *subst*[] characters:

```
if (t = strchr(keystr, *s))
    *s = subst[t - keystr];
```

## See Also

*memchr*, *strcspn*, *strpbrk*, *strchr*, *strspn*

## Notes

*strchr* is packaged in the integer library.

# strcmp

## Description

Compare two strings for lexical order

## Syntax

```
#include <string.h>
int strcmp(char *s1, char *s2)
```

## Function

**strcmp** compares two text strings, character by character, for lexical order in the character collating sequence. The first string starts at **s1**, the second at **s2**. The strings must match, including their terminating NUL characters, in order for them to be equal.

## Return Value

*strcmp* returns an integer greater than, equal to, or less than zero, according to whether **s1** is lexicographically greater than, equal to, or less than **s2**.

## Example

To look for the string “*include*”:

```
if (strcmp(buf, "include") == 0)
    doinclude();
```

## See Also

*memcmp*, *strncmp*

## Notes

*strcmp* is packaged in the integer library.

# strcpy

## Description

Copy one string to another

## Syntax

```
#include <string.h>
char *strcpy(char *s1, char *s2)
```

## Function

**strcpy** copies the NUL terminated string at **s2** to the buffer pointed at by **s1**. The terminating NUL is also copied.

## Return Value

*strcpy* returns **s1**.

## Example

To make a copy of the string **s2** in **dest**:

```
strcpy(dest, s2);
```

## See Also

*memcpy*, *strncpy*

## Notes

There is no way to specify the size of the destination area, to prevent storage overwrites.

*strcpy* is packaged in the `string` library, and may be implemented as an *inline* function.

# strcspn

## Description

Find the end of a span of characters in a set

## Syntax

```
#include <string.h>
unsigned int strcspn(char *s1, char *s2)
```

## Function

**strcspn** scans the string starting at **s1** for the first occurrence of a character in the string starting at **s2**. It computes a subscript **i** such that:

- **s1[i]** is a character in the string starting at **s1**
- **s1[i]** compares equal to some character in the string starting at **s2**, which may be its terminating null character.

## Return Value

*strcspn* returns the lowest possible value of **i**. **s1[i]** designates the terminating null character if none of the characters in **s1** are in **s2**.

## Example

To find the start of a decimal constant in a text string:

```
if (!str[i = strcspn(str, "0123456789+-")])
    printf("can't find number\n");
```

## See Also

*memchr*, *strchr*, *strpbrk*, *strrchr*, *strspn*

## Notes

*strcspn* is packaged in the integer library.

# strlen

## Description

Find length of a string

## Syntax

```
#include <string.h>
unsigned int strlen(char *s)
```

## Function

**strlen** scans the text string starting at **s** to determine the number of characters before the terminating NUL.

## Return Value

The value returned is the number of characters in the string before the NUL.

## Notes

*strlen* is packaged in the integer library and may be implemented as an *inline* function.



# strncat

## Description

Concatenate strings of length *n*

## Syntax

```
#include <string.h>
char *strncat(char *s1, char *s2, unsigned char n)
```

## Function

**strncat** appends a copy of the NUL terminated string at **s2** to the end of the NUL terminated string at **s1**. The first character of **s2** overlaps the NUL at the end of **s1**. **n** specifies the maximum number of characters to be copied, unless the terminating NUL in **s2** is encountered first. A terminating NUL is always appended to **s1**.

## Return Value

*strncat* returns **s1**.

## Example

To concatenate the strings “*day*” and “*light*”:

```
strcpy(s, "day");
strncat(s + 3, "light", 5);
```

## See Also

*strcat*

## Notes

*strncat* is packaged in the integer library.

# strncmp

## Description

Compare two *n* length strings for lexical order

## Syntax

```
#include <string.h>
int strncmp(char *s1, char *s2, unsigned char n)
```

## Function

**strncmp** compares two text strings, character by character, for lexical order in the character collating sequence. The first string starts at **s1**, the second at **s2**. **n** specifies the maximum number of characters to be compared, unless the terminating NUL in **s1** or **s2** is encountered first. The strings must match, including their terminating NUL character, in order for them to be equal.

## Return Value

*strncmp* returns an integer greater than, equal to, or less than zero, according to whether **s1** is lexicographically greater than, equal to, or less than **s2**.

## Example

To check for a particular error message:

```
if (strncmp(errmsg,
            "can't write output file", 23) == 0)
    cleanup(errmsg);
```

## See Also

*memcmp*, *strcmp*

## Notes

*strncmp* is packaged in the integer library.

# strncpy

## Description

Copy *n* length string

## Syntax

```
#include <string.h>
char *strncpy(char *s1, char *s2, unsigned char n)
```

## Function

**strncpy** copies the first **n** characters starting at location **s2** into the buffer beginning at **s1**. **n** specifies the maximum number of characters to be copied, unless the terminating NUL in **s2** is encountered first. In that case, additional NUL padding is appended to **s2** to copy a total of **n** characters.

## Return Value

*strncpy* returns **s1**.

## Example

To make a copy of the string **s2** in *dest*:

```
strncpy(dest, s2, n);
```

## See Also

*memcpy*, *strcpy*

## Notes

If the string **s2** points at is longer than **n** characters, the result may not be NUL-terminated.

*strncpy* is packaged in the integer library.

# strpbrk

## Description

Find occurrence in string of character in set

## Syntax

```
#include <string.h>
char *strpbrk(char *s1, char *s2)
```

## Function

**strpbrk** scans the NUL terminated string starting at **s1** for the first occurrence of a character in the NUL terminated set **s2**.

## Return Value

*strpbrk* returns a pointer to the first character in **s1** that is also contained in the set **s2**, or a NULL if none does.

## Example

To replace unprintable characters (as for a 64 character terminal):

```
while (string = strpbrk(string, "\\{|}~"))
    *string = '@';
```

## See Also

*memchr*, *strchr*, *strcspn*, *strrchr*, *strspn*

## Notes

*strpbrk* is packaged in the integer library.

# strchr

## Description

Scan string for last occurrence of character

## Syntax

```
#include <string.h>
char *strrchr(char *s, char c)
```

## Function

**strrchr** looks for the last occurrence of a specific character **c** in a NUL terminated string starting at **s**.

## Return Value

*strrchr* returns a pointer to the last character that matches **c**, or NULL if none does.

## Example

To find a filename within a directory pathname:

```
if (s = strrchr("/usr/lib/libc.user", '/'))
    ++s;
```

## See Also

*memchr*, *strchr*, *strpbrk*, *strcspn*, *strspn*

## Notes

*strchr* is packaged in the integer library.

# strspn

## Description

Find the end of a span of characters not in set

## Syntax

```
#include <string.h>
unsigned int strspn(char *s1, char *s2)
```

## Function

**strspn** scans the string starting at **s1** for the first occurrence of a character not in the string starting at **s2**. It computes a subscript **i** such that

- **s1[i]** is a character in the string starting at **s1**
- **s1[i]** compares equal to no character in the string starting at **s2**, except possibly its terminating null character.

## Return Value

*strspn* returns the lowest possible value of **i**. **s1[i]** designates the terminating null character if all of the characters in **s1** are in **s2**.

## Example

To check a string for characters other than decimal digits:

```
if (str[strspn(str, "0123456789")])
    printf("invalid number\n");
```

## See Also

*memchr, strcspn, strchr, strpbrk, strrchr*

## Notes

*strspn* is packaged in the integer library.

# strstr

## Description

Scan string for first occurrence of string

## Syntax

```
#include <string.h>
char *strstr(char *s1, char *s2)
```

## Function

**strstr** looks for the first occurrence of a specific string **s2** not including its terminating NUL, in a NUL terminated target string **s1**.

## Return Value

*strstr* returns a pointer to the first character that matches **c**, or NULL if none does.

## Example

To look for a keyword in a string:

```
if (t = strstr(buf, "LIST"))
    do_list(t);
```

## See Also

*memchr*, *strcspn*, *strpbrk*, *strchr*, *strspn*

## Notes

*strstr* is packaged in the integer library.

# strtod

## Description

Convert buffer to double

## Syntax

```
#include <stdlib.h>
double strtod(char *nptr, char **endptr)
```

## Function

**strtod** converts the string at *nptr* into a double. The string is taken as the text representation of a decimal number, with an optional fraction and exponent. Leading whitespace is skipped and an optional sign is permitted; conversion stops on the first unrecognizable character. Acceptable inputs match the pattern:

$$[+|-]d*.[d*][e[+|-]dd*]$$

where **d** is any decimal digit and **e** is the character ‘e’ or ‘E’. If *endptr* is not a null pointer, *\*endptr* is set to the address of the first unconverted character remaining in the string *nptr*. No checks are made against overflow, underflow, or invalid character strings.

## Return Value

*strtod* returns the converted double value. If the string has no recognizable characters, it returns zero.

## Example

To read a string from STDIN and convert it to a double at **d**:

```
gets(buf);
d = strtod(buf, NULL);
```

## See Also

*atoi*, *atol*, *strtol*, *strtoul*

## Notes

*strtod* is packaged in the floating point library.



# strtol

## Description

Convert buffer to long

## Syntax

```
#include <stdlib.h>
long strtol(char *nptr, char **endptr, char base)
```

## Function

**strtol** converts the string at *nptr* into a long integer. Leading whitespace is skipped and an optional sign is permitted; conversion stops on the first unrecognizable character. If base is not zero, characters **a-z** or **A-Z** represents digits in range 10-36. If base is zero, a leading **"0x"** or **"0X"** in the string indicates hexadecimal, a leading **"0"** indicates octal, otherwise the string is take as a decimal representation. If base is 16 and a leading **"0x"** or **"0X"** is present, it is skipped before to convert. If *endptr* is not a null pointer, *\*endptr* is set to the address of the first unconverted character in the string *nptr*.

No checks are made against overflow or invalid character strings.

## Return Value

*strtol* returns the converted long integer. If the string has no recognizable characters, zero is returned.

## Example

To read a string from STDIN and convert it to a long I:

```
gets(buf);
l = strtol(buf, NULL, 0);
```

## See Also

*atof*, *atoi*, *strtoul*, *strtod*

## Notes

*strtol* is packaged in the integer library.

# strtoul

## Description

Convert buffer to unsigned long

## Syntax

```
#include <stdlib.h>
unsigned long strtoul(char *nptr, char **endptr,
                     char base)
```

## Function

**strtoul** converts the string at *nptr* into a long integer. Leading whitespace is skipped and an optional sign is permitted; conversion stops on the first unrecognizable character. If base is not zero, characters **a-z** or **A-Z** represents digits in range 10-36. If base is zero, a leading “**0x**” or “**0X**” in the string indicates hexadecimal, a leading “**0**” indicates octal, otherwise the string is take as a decimal representation. If base is 16 and a leading “**0x**” or “**0X**” is present, it is skipped before to convert. If *endptr* is not a null pointer, *\*endptr* is set to the address of the first unconverted character in the string *nptr*.

No checks are made against overflow or invalid character strings.

## Return Value

*strtoul* returns the converted long integer. If the string has no recognizable characters, zero is returned.

## Example

To read a string from STDIN and convert it to a long **l**:

```
gets(buf);
l = strtoul(buf, NULL, 0);
```

## See Also

*atof*, *atoi*, *strtol*, *strtod*

## Notes

*strtoul* is a macro redefined to *strtol*.

# tan

## Description

Tangent

## Syntax

```
#include <math.h>
double tan(double x)
```

## Function

**tan** computes the tangent of **x**, expressed in radians, to full double precision.

## Return Value

*tan* returns the nearest internal representation to  $\tan(x)$ , in the range  $[-\pi/2, \pi/2]$ , expressed as a double floating value. If the number in **x** is too large to be represented, **tan** returns zero. An argument with a large size may return a meaningless value, *i.e.* when **x**/**(2 \* pi)** has no fraction bits.

## Example

To compute the tangent of *theta*:

```
y = tan(theta);
```

## See Also

*cos*, *sin*

## Notes

*tan* is packaged in the floating point library.

# tanh

## Description

Hyperbolic tangent

## Syntax

```
#include <math.h>
double tanh(double x)
```

## Function

**tanh** computes the value of the hyperbolic tangent of *x* to double precision.

## Return Value

*tanh* returns the nearest internal representation to *tanh(x)*, expressed as a double floating value. If the result is too large to be properly represented, *tanh* returns zero.

## Example

To compute the hyperbolic tangent of *x*:

```
y = tanh(x);
```

## See Also

*cosh*, *exp*, *sinh*

## Notes

*tanh* is packaged in the floating point library.

# tolower

## Description

Convert character to lowercase if necessary

## Syntax

```
#include <ctype.h>
int tolower(char c)
```

## Function

**tolower** converts an uppercase letter to its lowercase equivalent, leaving all other characters unmodified.

## Return Value

*tolower* returns the corresponding lowercase letter, or the unchanged character.

## Example

To accumulate a hexadecimal digit:

```
for (sum = 0; isxdigit(*s); ++s)
    if (isdigit(*s))
        sum = sum * 16 + *s - '0';
    else
        sum = sum * 16 + tolower(*s) + (10 - 'a');
```

## See Also

*toupper*

## Notes

*tolower* is packaged in the integer library.

# toupper

## Description

Convert character to uppercase if necessary

## Syntax

```
#include <ctype.h>
int toupper(char c)
```

## Function

**toupper** converts a lowercase letter to its uppercase equivalent, leaving all other characters unmodified.

## Return Value

*toupper* returns the corresponding uppercase letter, or the unchanged character.

## Example

To convert a character string to uppercase letters:

```
for (i = 0; i < size; ++i)
    buf[i] = toupper(buf[i]);
```

## See Also

*tolower*

## Notes

*toupper* is packaged in the integer library.

# Using The Assembler

---

The **castm8** cross assembler translates your assembly language source files into relocatable object files. The C cross compiler calls *castm8* to assemble your code automatically, unless specified otherwise. *castm8* generates also listings if requested. This chapter includes the following sections:

- Invoking castm8
- Object File
- Listings
- Assembly Language Syntax
- Branch Optimization
- Old Syntax
- C Style Directives
- Assembler Directives

## Invoking *castm8*

*castm8* accepts the following command line options, each of which is described in detail below:

```
castm8 [options] <files>
-a    absolute assembler
-b    do not optimize branches
-c    output cross reference
-d*>  define symbol=value
+e*   error file name
-ff   use formfeed in listing
-ft   force title in listing
-f#   fill byte value
-ge   eclipse error messages
-h*   include header
-i*>  include path
-l    output listing
+l*   listing file name
-m    accept old syntax
-md   make dependencies
-mi   accept label syntax
-o*   output file name
-pe   all equates public
-p    all symbols public
-pl   keep local symbol
-si   suppress .info. section
-u    undefined in listing
-v    be verbose
-x    include line debug info
-xp   no path in debug info
-xx   include full debug info
```

### Castm8 Option Usage

Option	Description
<b>-a</b>	map all sections to absolute, including the predefined ones.



## Castm8 Option Usage (cont.)

Option	Description
<b>-b</b>	do not optimize branch instructions. By default, the assembler replaces long branches by short branches wherever a shorter instruction can be used, and short branches by long branches wherever the displacement is too large. This optimization also applies to jump and jump to subroutines instructions.
<b>-c</b>	produce cross-reference information. The cross-reference information will be added at the end of the listing file. This option enforces the <b>-l</b> option.
<b>-d*&gt;</b>	where * has the form <b>name=value</b> , defines <b>name</b> to have the value specified by <b>value</b> . This option is equivalent to using an <b>equ</b> directive in each of the source files.
<b>+e*</b>	log errors from assembler in the text file * instead of displaying the messages on the terminal screen.
<b>-ff</b>	use <i>formfeed</i> character to skip pages in listing instead of using blank lines.
<b>-ft</b>	output a title in listing (date, file name, page). By default, no title is output.
<b>-f#</b>	define the value of the filling byte used to fill any gap created by the assembler directives. Default is 0.
<b>-ge</b>	produce error messages directly compatible with the Eclipse environment.
<b>-h*</b>	include the file specified by * before starting assembly. It is equivalent to an <b>include</b> directive in each source file.
<b>-i*&gt;</b>	define a path to be used by the <b>include</b> directive. Up to 128 paths can be defined. A path is a directory name and is <b>not</b> ended by any directory separator character, or a file containing an unlimited list of directory names.
<b>-l</b>	create a listing file. The name of the listing file is derived from the input file name by replacing the suffix by the <b>‘.js’</b> extension, unless the <b>+l</b> option has been specified.
<b>+l*</b>	create a listing file in the text file *. If both <b>-l</b> and <b>+l</b> are specified, the listing file name is given by the <b>+l</b> option.

## Castm8 Option Usage (cont.)

Option	Description
<b>-m</b>	accept the old syntax.
<b>-md</b>	create only a list of 'make' compatible dependencies consisting for each source file in the object name followed by a list of included files needed to assemble that file.
<b>-mi</b>	accept label that is not ended with a ':' character.
<b>-o*</b>	write object code to the file *. If no file name is specified, the output file name is derived from the input file name, by replacing the rightmost extension in the input file name with the character 'o'. For example, if the input file name is <i>prog.s</i> , the default output file name is <i>prog.o</i> .
<b>-pe</b>	mark all symbols defined by an <b>equ</b> directive as <b>public</b> . This option has the same effect than adding a <b>xdef</b> directive for each of those symbols.
<b>-pl</b>	put locals in the symbol table. They are not published as externals and will be only displayed in the linker map file.
<b>-p</b>	mark all defined symbols as <b>public</b> . This option has the same effect than adding a <b>xdef</b> directive for each label.
<b>-si</b>	suppress the <b>.info</b> section produced automatically and containing the object name, date and assembler options.
<b>-u</b>	produce an error message in the listing file for all occurrence of an undefined symbol. This option enforces the <b>-l</b> option.
<b>-v</b>	display the name of each file which is processed.
<b>-x</b>	add line debug information to the object file.
<b>-xp</b>	do not prefix filenames in the debug information with any absolute path name. Debuggers will have to be informed about the actual files location.
<b>-xx</b>	add debug information in the object file for any label defining code or data. This option disables the <b>-p</b> option as only public or used labels are selected.

Each source file specified by `<files>` will be assembled separately, and will produce separate object and listing files. For each source file, if no errors are detected, *castm8* generates an object file. If requested by the **-l** or **-c** options, *castm8* generates a listing file even if errors are detected. Such lines are followed by an error message in the listing.

## Object File

The object file produced by the assembler is a relocatable object in a format suitable for the linker *clnk*. This will normally consist of machine code, initialized data and relocation information. The object file also contains information about the sections used, a symbol table, and a debug symbol table.

The object file contains by default a special section named **.info**, containing text strings. The assembler produces the object name followed by the current date and time, and the assembler name followed by the specified options. The other compiler passes will complete this information to add their own name and options. This section can be suppressed for compatibility reason by using the **-si** option.

## Listings

The listing stream contains the source code used as input to the assembler, together with the hexadecimal representation of the corresponding object code and the address for which it was generated. The contents of the listing stream depends on the occurrence of the **list**, **nolist**, **clist**, **dlist** and **mlist** directives in the source. The format of the output is as follows:

`<address> <generated_code> <source_line>`

where `<address>` is the hexadecimal relocatable address where the `<source_line>` has been assembled, `<generated_code>` is the hexadecimal representation of the object code generated by the assembler and `<source_line>` is the original source line input to the assembler. If expansion of data, macros and included files is not enabled, the `<generated_code>` print will not contain a complete listing of all generated code.

Addresses in the listing output are the offsets from the start of the current section. After the linker has been executed, the listing files may be updated to contain absolute information by the **clabs** utility. Addresses and code will be updated to reflect the actual values as built by the linker.

Several directives are available to modify the listing display, such as **title** for the page header, **plen** for the page length, **page** for starting a new page, **tabs** for the tabulation characters expansion. By default, the listing file is not paginated. Pagination is enabled by using at least one **title** directive in the source file, or by specifying the **-ft** option on the command line. Otherwise, the **plen** and **page** directives are simply ignored. Some other directives such as **clist**, **mlist** or **dlist** control the amount of information produced in the listing.

A **cross-reference** table will be appended to the listing if the **-c** option has been specified. This table gives for each symbol its value, its attributes, the line number of the line where it has been defined, and the list of line numbers where it is referenced.

## Assembly Language Syntax

The assembler *castm8* conforms to the STM8 syntax as described in the document *Assembly Language Input Standard*. The assembly language consists of lines of text in the form:

```
[label:] [command [operands]] [; comment]
or
; comment
```

where ‘:’ indicates the end of a label and ‘;’ defines the start of a comment. The end of a line terminates a comment. The *command* field may be an **instruction**, a **directive** or a **macro call**.

Instruction mnemonics and assembler directives may be written in upper or lower case. The C compiler generates lowercase assembly language.

A source file must end with the **end** directive. All the following lines will be ignored by the assembler. If an **end** directive is found in an included file, it stops only the process for the included file.

## Instructions

*castm8* recognizes the following instructions:

adc	ccf	incw	jrn	ldf	rim	sra
add	clr	iret	jrn	ldw	rlc	sraw
addw	clrw	jp	jrn	mov	rlcw	srl
and	cp	jpf	jrpl	mul	rlwa	srlw
bccm	cpl	jra	jrsg	neg	rrc	sub
bcp	cplw	jrc	jrsgt	negw	rrcw	subw
bcpl	cpw	jreq	jrsl	nop	rrwa	swap
bkpt	dec	jrf	jrslt	or	rvf	swapw
bres	decw	jr	jrt	pop	sbc	tnz
bset	div	jri	jrge	popw	scf	tnzw
btjf	divw	jr	jrgt	push	sim	trap
btjt	exg	jrm	jrle	pushw	sla	wfe
call	exgw	jrm	jrult	rcf	slaw	wfi
callf	halt	jrnc	jr	ret	sll	xor
callr	inc	jrne	ld	retf	sllw	

The **operand** field of an instruction uses an addressing mode to describe the instruction argument. The following examples demonstrate the accepted syntax:

```
rcf                ; implicit
push y             ; register
ld a,#1            ; immediate
and a,var           ; short, long
ld a,(2,x)          ; indexed
ld a,[var]          ; indirect
ld a,([var.w],y)    ; indirect indexed long
jrne loop           ; relative
bset 2,#1           ; bit number
btjt 2,#1,loop      ; bit test and branch
```

The assembler chooses the smallest addressing mode where several solutions are possible. *Short* addressing mode is selected when using a label defined in the *.bsct* section.

For an exact description of the above instructions, refer to the ST Microelectronics's *STM8 Family Programming Manual*.

## Labels

A source line may begin with a label. Some directives require a label on the same line, otherwise this field is optional. A label must begin with

an alphabetic character, the underscore character ‘\_’ or the period character ‘.’. It is continued by alphabetic (A-Z or a-z) or numeric (0-9) characters, underscores, dollar signs (\$) or periods. Labels are case sensitive. The processor register names ‘*a*’ and ‘*x*’ are reserved and cannot be used as labels.

```
data1: dc.b      $56
c_x:   ds.b      1
```

When a label is used within a macro, it may be expanded more than once and in that case, the assembler will fail with a *multiply defined symbol* error. In order to avoid that problem, the special sequence ‘\@’ may be used as a label prefix. This sequence will be replaced by a unique sequence for each macro expansion. This prefix is only allowed inside a macro definition.

```
wait:  macro
\@loop:btjf    PORTA,#1,\@loop
      endm
```

## Temporary Labels

The assembler allows temporary labels to be defined when there is no need to give them an explicit name. Such a label is composed by a decimal number immediately followed by a ‘\$’ character. Such a label is valid until the next standard label or the *local* directive. Then the same temporary label may be redefined without getting a multiply defined error message.

```
1$:    dec
      jrne 1$
2$:    dec
      jrne 2$
```

Temporary labels do not appear in the symbol table or the cross reference list.

For example, to define 3 different local blocks and create and use 3 different local labels named 10\$:

```
function1:
10$:    ld      a,var
      jreq     10$
```

```

        ld      a,var2
local
10$:    ld      a,var2
        jreq    10$
        ld      a,var
        ret
function2:
10$:    ld      a,var2
        sub     a,var
        jrne    10$
        ret

```

## Constants

The assembler accepts **numeric** constants and **string** constants. *Numeric* constants are expressed in different bases depending on a *prefix* character as follows:

Number	Base
<b>10</b>	decimal (no prefix)
<b>%1010</b>	binary
<b>@12</b>	octal
<b>\$1F or 0x1F or 0X1F</b>	hexadecimal

The assembler also accepts numerics constants in different bases depending on a *suffix* character as follow:

Suffix	Base
<b>D, d or none</b>	decimal (no prefix)
<b>B or b</b>	binary
<b>Q or q</b>	octal
<b>0AH or 0Ah</b>	hexadecimal

The suffix letter can be entered uppcase or lowercase. Hexadecimal numbers still need to start with a digit.

*String* constants are a series of printable characters between single or double quote characters:

```
'This is a string'
"This is also a string"
```

Depending on the context, a string constant will be seen either as a series of bytes, for a data initialization, or as a numeric; in which case, the string constant should be reduced to only one character.

```
hexa:  dc.b      '0123456789ABCDEF'
start:  cp        x, #'A'    ; ASCII value of 'A'
```

## Expressions

An expression consists of a number of labels and constants connected together by operators. Expressions are evaluated to 32-bit precision. Note that operators have the same precedence than in the C language.

A special label written **\*\*** is used to represent the current location address. Note that when **\*\*** is used as the operand of an instruction, it has the value of the program counter **before** code generation for that instruction. The set of accepted operators is:

+	addition
-	subtraction (negation)
*	multiplication
/	division
%	remainder (modulus)
&	bitwise and
	bitwise or
^	bitwise exclusive or
~	bitwise complement
<<	left shift
>>	right shift
==	equality
!=	difference
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
&&	logical and
	logical or
!	logical complement



These operators may be applied to constants without restrictions, but are restricted when applied to *relocatable* labels. For those labels, the **addition** and **subtraction** operators only are accepted and only in the following cases:

```
label + constant
label - constant
label1 - label2
```

### NOTE

*The difference of two relocatable labels is valid only if both symbols are not external symbols, and are defined in the same section.*

An expression may also be constructed with a special operator. These expressions cannot be used with the previous operators and have to be specified alone.

<b>high(expression)</b>	upper byte
<b>low(expression)</b>	lower byte
<b>page(expression)</b>	page byte

These special operators evaluate an **expression** and extract the appropriate information from the result. The expression may be relocatable, and may use the set of operators if allowed.

**high** - extract the upper byte of the 16-bit expression

**low** - extract the lower byte of the 16-bit expression

**page** - extract the *page* value of the expression. It is computed by the linker according to the **-bs** option used. This is used to get the address extension when bank switching is used.

## Macro Instructions

A **macro** instruction is a list of assembler commands collected under a unique name. This name becomes a new command for the following of the program. A **macro** begins with a **macro** directive and ends with a **endm** directive. All the lines between these two directives are recorded and associated with the macro name specified with the **macro** directive.

```

signex:macro                ; sign extension
    clr     x                ; prepare MSB
    tnz     a                ; test sign
    jrpl    \@pos            ; if not positive
    cpl     x                ; invert MSB
\@pos:
    endm                    ; end of macro

```

This macro is named *signex* and contains the code needed to perform a sign extension of **a** into **x**. Whenever needed, this macro can be expanded just by using its name in place of a standard instruction:

```

ld     a,char+1; load LSB
signex          ; expand macro
ld     char,x   ; store result

```

The resulting code will be the same as if the following code had been written:

```

ld     a,char+1; load LSB
clr     x                ; prepare MSB
tnz     a                ; test sign
jrpl    pos            ; if not positive
cpl     x                ; invert MSB
pos:
    ld     char,x   ; store result

```

A **macro** may have up to 35 *parameters*. A *parameter* is written **\1**, **\2**,... **\9**, **\A**,...**\Z** inside the macro body and refers explicitly to the first, second,... ninth *argument* and **\A** to **\Z** to denote the tenth to 35th operand on the invocation line, which are placed after the macro name, and separated by commas. Each *argument* replaces each occurrence of its corresponding *parameter*. An *argument* may be expressed as a **string** constant if it contains a comma character.

A macro can also handle named arguments instead of numbered argument. In such a case, the macro directive is followed by a list of argument named, each prefixed by a **\** character, and separated by commas. Inside the macro body, arguments will be specified using the same syntax or a sequence starting by a **\** character followed by the argument named placed between parenthesis. This alternate syntax is useful to concatenate the argument with a text string immediately starting with alphanumeric characters.

The special *parameter* `\#` is replaced by a numeric value corresponding to the number of *arguments* actually found on the invocation line.

In order to operate directly in memory, the previous macro may have been written using the **numbered** syntax:

```
signex:macro                ; sign extension
    clr    x                ; prepare MSB
    ld     a,\1             ; load LSB
    jrpl   \@pos            ; if not positive
    cpl    x                ; invert MSB
\@pos:
    ld     \1,x             ; store MSB
endm                      ; end of macro
```

And called:

```
signex char                ; sign extend char
```

This macro may also be written using the **named** syntax:

```
signex:macro    \value      ; sign extension
    clr    x                ; prepare MSB
    ld     a,\value         ; load LSB
    jrpl   \@pos            ; if not positive
    cpl    x                ; invert MSB
\@pos:
    ld     \ (value) ,x     ; store MSB
endm            ; end of macro
```

The form of a macro call is:

<b>name&gt;[.&lt;ext&gt;] [&lt;arguments&gt;]</b>
---

The special parameter `\0` corresponds to an extension `<ext>` which may follow the macro name, separated by the period character `'.'`. An extension is a single letter which may represent the size of the operands and the result. For example:

```
table: macro
    dc.\0    1,2,3,4
endm
```

When invoking the macro:

**table.b**

will generate a table of byte:

```
dc.b      1, 2, 3, 4
```

When invoking the macro:

**table.w**

will generate a table of word:

```
dc.w      1, 2, 3, 4
```

The special parameter **\\*** is replaced by a sequence containing the list of all the passed arguments separated by commas. This syntax is useful to pass all the macro arguments to another macro or a **repeatl** directive.

The directive **mexit** may be used at any time to stop the macro expansion. It is generally used in conjunction with a conditional directive.

A macro call may be used within another macro definition, all macros must then be defined before their first call. A macro definition cannot contain another macro definition.

If a listing is produced, the macro expansion lines are printed if enabled by the **mlist** directive. If enabled, the invocation line is not printed, and all the expanded lines are printed with all the *parameters* replaced by their corresponding *arguments*. Otherwise, the invocation line only is printed.

## Conditional Directives

A **conditional directive** allows parts of the program to be assembled or not depending on a specific condition expressed in an **if** directive. The condition is an expression following the **if** command. The expression cannot be relocatable, and shall evaluate to a numeric result. If the condition is *false* (expression evaluated to zero), the lines following the **if** directive are skipped until an **endif** or **else** directive. Otherwise, the lines are normally assembled. If an **else** directive is encountered, the condition status is reversed, and the conditional process continues until the next **endif** directive.

```

if          debug == 1
ld           x, #message
call        print
endif

```

If the symbol `debug` is equal to 1, the next two lines are assembled. Otherwise they are skipped.

```

if          offset != 1      ; if offset too large
addptr      offset          ; call a macro
else
inc          x              ; otherwise
endif          increment X register

```

If the symbol `offset` is not one, the macro `addptr` is expanded with `offset` as argument, otherwise the `inc` instruction is directly assembled.

Conditional directives may be nested. An **else** directive refers to the closest previous **if** directive, and an **endif** directive refers to the closest previous **if** or **else** directive.

If a listing is produced, the skipped lines are printed only if enabled by the **clist** directive. Otherwise, only the assembled lines are printed.

## Sections

The assembler allows code and data to be splitted in **sections**. A *section* is a set of code or data referenced by a section name, and providing a contiguous block of relocatable information. A *section* is defined with a *section* directive, which creates a new section and redirects the following code and data thereto. The directive **switch** can be used to redirect the following code and data to another *section*.

```

data:  section           ; defines data section
text:  section           ; defines text section
start:
ld       x, #value; fills text section
jp       print
switch data      ; use now data section
value:
dc.b     1,2,3      ; fills data section

```

The assembler allows up to **255** different sections. A section name is limited to **15** characters. If a section name is too long, it is simply truncated without any error message.

The assembler predefines the following sections, meaning that a *section* directive is *not* needed before to use them:

Section	Description
<b>.text</b>	executable code
<b>.data</b>	initialized data
<b>.bss</b>	uninitialized data
<b>.bsct</b>	initialized data in short range
<b>.ubsct</b>	non initialized data in short range-short range

The sections **.bsct** and **.ubsct** are used for locating data in the short range of the processor. The short range is defined as the memory addresses between **0x00** and **0xFF** inclusive, i.e. the memory directly addressable by a single byte. Several processors include special instructions and/or addressing modes that take advantage of this special address range. The Cosmic assembler will automatically use the most efficient addressing mode if the data objects are allocated in the **.bsct**, **.ubsct** or a section with the same attributes. If short range data objects are defined in another file then the directive **xref.b** must be used to externally reference the data object. This directive specifies that the address for these data object is only one byte and therefore the assembler may use 8 bit addressing modes.

```
        switch .bsct
zvar2:  ds.b   1
        switch .bss
var2:   ds.b   1
        switch .text
        ld     a,var
        ld     a,zvar
        ld     a,var2
        ld     a,var2
        end
```

## Bit Handling

The assembler allows symbols specifying bit addresses instead of byte addresses. A bit address is obtained from a byte address and a bit number by 'ing the bit number with the byte address 3-bit shifted to

the left. Such symbol can be defined either by an equate definition or as member of a bit section. Such a section can be defined by using the **section** directive with the **bit** attribute. In a bit section, any directive creating or reserving bytes can be used, but will create or reserve bits. Bit symbols can be directly used by the bit instructions with a shortened syntax, as a bit symbol is defining both a byte and a bit in this byte. Bit symbols can be declared as external by using the **xbit** directive. External definitions for bit symbols located in the short range will use the **xbit.b** directive.

```

        xbit.b  b1          ; external bit declaration
PA3:    equ     PORTA:3    ; bit 3 of byte PORTA
.bit:   section  zpage,bit; create bit section named ".bit"
b0:     ds.b     1          ; allocates one bit
        switch  .text
        btjfb   PA3,skip; use directly bit symbol
        bset    b0          ; with bit instructions
skip:
        bres    b1

```

Bit sections are located at link time either at specified bit addresses or attached to any short range section. The linker is computing the proper addresses when hooking bit sections to byte sections, or byte sections to bit sections.

## Includes

The **include** directive specifies a file to be included and assembled in place of the **include** directive. The file name is written between double quotes, and may be any character string describing a file on the host system. If the file cannot be found using the given name, it is searched from all the include paths defined by the **-i** options on the command line, and from the paths defined by the environment symbol **CXLIB**, if such a symbol has been defined before the assembler invocation. This symbol may contain several paths separated by the usual path separator of the host operating system (‘;’ for MSDOS and ‘:’ for UNIX).

The **-h** option can specify a file to be “included”. The file specified will be included as if the program had an **include** directive at its very top. The specified file will be included before **any** source file specified on the command line.

## Branch Optimization

Branch instructions are by default automatically optimized to produce the shortest code possible. This behaviour may be disabled by the **-b** option. This optimization operates on conditional branches, on jumps and jumps to subroutine.

A *conditional* branch offset is limited to the range **[-128,127]**. If such an instruction cannot be encoded properly, the assembler will replace it by a sequence containing an inverted branch to the next location followed immediately by a jump to the original target address. The assembler keep track of the last replacement for each label, so if a long branch has already been expanded for the same label at a location close enough from the current instruction, the target address of the short branch will be changed only to branch on the already existing jump instruction to the specified label.

```
jreq    farlabel    becomes    jrne    *+5
                                jp      farlabel
```

Note that a *jra* instruction will be replaced by a single *jp* instruction if it cannot be encoded as a relative branch.

A *jp* or *call* instruction will be replaced by a *jra* or *callr* instruction if the destination address is in the same section than the current one, and if the displacement is in the range allowed by a relative branch.

## Old Syntax

The **-m** option allows the assembler to accept old constructs which are now obsolete. The following features are added to the standard behaviour:

- a comment line may begin with a ‘**\***’ character;
- a label starting in the first column does not need to be ended with a ‘**:**’ character;
- no error message is issued if an operand of the **dc.b** directive is too large;
- the **section** directive handles *numbered* sections;



The comment separator at the end of an instruction is still the ‘;’ character because the ‘\*’ character is interpreted as the multiply operator.

## C Style Directives

The assembler also supports C style directives matching the preprocessor directives of a C compiler. The following directives list shows the equivalence with the standard directives:

C Style	Assembler Style
#include “file”	include “file”
#define label expression	label: equ expression
#define label	label: equ 1
#if expression	if expression
#ifdef label	ifdef label
#ifndef label	ifndef label
#else	else
#endif	endif
#error “message”	fail “message”

### NOTE

*The #define directive does not implement all the text replacement features provided by a C compiler. It can be used only to define a symbol equal to a numerical value.*

## Assembler Directives

This section consists of quick reference descriptions for each of the *castm8* assembler directives.

# align

## Description

Align the next instruction on a given boundary

## Syntax

```
align <expression>, [<fill_value>]
```

## Function

The **align** directive forces the next instruction to start on a specific boundary. The **align** directive is followed by a constant expression which must be positive. The next instruction will start at the next address which is a multiple of the specified value. If bytes are added in the section, they are set to the value of the filling byte defined by the **-f** option. If **<fill\_value>** is specified, it will be used locally as the filling byte, instead of the one specified by the **-f** option.

## Example

```
align    3           ; next address is multiple of 3
ds.b     1
```

## See Also

*even*

# base

## Description

Define the default base for numerical constants

## Syntax

```
base <expression>
```

## Function

The **base** directive sets the default base for numerical constants beginning with a digit. The **base** directive is followed by a constant expression which value must be one of **2**, **8**, **10** or **16**. The decimal base is used by default. When another base is selected, it is no more possible to enter decimal constants.

## Example

```
base      8                ; select octal base
ld        a, #377          ; load $FF
```

# bsct

## Description

Switch to the predefined **.bsct** section.

## Syntax

<b>bsct</b>
-------------

## Function

The **bsct** directive switches input to a section named **.bsct**, also known as the **short range** section. The assembler will automatically select the *short* addressing mode when referencing an object defined in the *.bsct* section.

## Example

```
        bsct
c_x:
        ds.b      1
```

## Notes

The *.bsct* section is limited to 256 bytes, but the assembler does not check the *.bsct* section size. This will be done by the linker.

## See Also

*section, switch*

# clist

## Description

Turn listing of conditionally excluded code on or off.

## Syntax

```
clist [on|off]
```

## Function

The **clist** directive controls the output in the listing file of conditionally excluded code. It is effective if and only if listings are requested; it is ignored otherwise.

The parts of the program to be listed are the program lines which are not assembled as a consequence of **if**, **else** and **endif** directives.

## See Also

*if, else, endif*

# dc

## Description

Allocate constant(s)

## Syntax

`dc[.size] <expression>[,<expression>...]`

## Function

The **dc** directive allocates and initializes storage for constants. If *<expression>* is a string constant, one byte is allocated for each character of the string. Initialization can be specified for each item by giving a series of values separated by commas or by using a repeat count.

The **dc** and **dc.b** directives will allocate one byte per *<expression>*.

The **dc.w** directive will allocate one word per *<expression>*.

The **dc.l** directive will allocate one long word per *<expression>*.

## Example

```
digit: dc.b    10,'0123456789'
       dc.w    digit
```

## Note

For compatibility with previous assemblers, the directive **fcbl** is alias to **dc.b**, and the directive **fdw** is alias to **dc.w**.

# dcb

## Description

Allocate constant block

## Syntax

```
dcb.<size> <count>,<value>
```

## Function

The **dcb** directive allocates a memory block and initializes storage for constants. The size area is the number of the specified value *<count>* of *<size>*. The memory area can be initialized with the *<value>* specified.

The **dcb** and **dcb.b** directives will allocate one **byte** per *<count>*.

The **dcb.w** directive will allocate one **word** per *<count>*.

The **dcb.l** directive will allocate one **long word** per *<count>*.

## Example

```
digit: dcb.b    10,5    ; allocate 10 bytes,  
                        ; all initialized to 5
```

# dlist

## Description

Turn listing of debug directives on or off.

## Syntax

```
dlist [on|off]
```

## Function

The **dlist** directive controls the visibility of any debug directives in the listing. It is effective if and only if listings are requested; it is ignored otherwise.



# ds

## Description

Allocate variable(s)

## Syntax

```
ds [.size] <space>
```

## Function

The **ds** directive allocates storage space for variables. <space> must be an absolute expression. Bytes created are set to the value of the filling byte defined by the **-f** option.

The **ds** and **ds.b** directives will allocate <space> bytes.

The **ds.w** directive will allocate <space> words.

The **ds.l** directive will allocate <space> long words.

## Example

```
ptlec:  ds.b    2
ptecr:  ds.b    2
chrbuf: ds.w   128
```

## Note

For compatibility with previous assemblers, the directive **rmb** is alias to **ds.b**.

# else

## Description

Conditional assembly

## Syntax

```
if <expression>
instructions
else
instructions
endc
```

## Function

The **else** directive follows an **if** directive to define an alternative conditional sequence. It reverts the condition status for the following instructions up to the next matching **endif** directive. An **else** directive applies to the closest previous **if** directive.

## Example

```
if      offset != 1      ; if offset too large
addptr  offset          ; call a macro
else    ; otherwise
inc     x               ; increment X register
endif
```

## Note

The **else** and **elsec** directives are equivalent and may be used without distinction. They are provided for compatibility with previous assemblers.

## See Also

*if, endif, clist*

# elsec

## Description

Conditional assembly

## Syntax

```
if <expression>
instructions
elsec
instructions
endc
```

## Function

The **elsec** directive follows an **if** directive to define an alternative conditional sequence. It reverts the condition status for the following instructions up to the next matching **endc** directive. An **elsec** directive applies to the closest previous **if** directive.

## Example

```
ifge      offset-127      ; if offset too large
addptr    offset          ; call a macro
elsec     ; otherwise
inc       x               ; increment X register
endc
```

## Note

The **elsec** and **else** directives are equivalent and may be used without distinction. They are provided for compatibility with previous assemblers.

## See Also

*if, endc, clist, else*

# end

## Description

Stop the assembly

## Syntax

<b>end</b>
------------

## Function

The **end** directive stops the assembly process. Any statements following it are ignored. If the **end** directive is encountered in an included file, it will stop the assembly process for the included file only.

# endc

## Description

End conditional assembly

## Syntax

```
if<cc> <expression>
instructions
endc
```

## Function

The **endc** directive closes an **if<cc>** or **elsec** conditional directive. The conditional status reverts to the one existing before entering the **if<cc>** directives. The **endc** directive applies to the closest previous **if<cc>** or **elsec** directive.

## Example

```
ifge      offset-127      ; if offset too large
addptr    offset          ; call a macro
elsec     ; otherwise
inc       x               ; increment X register
endc
```

## Note

The **endc** and **endif** directives are equivalent and may be used without distinction. They are provided for compatibility with previous assemblers.

## See Also

*if<cc>, elsec, clist, end*

# endif

## Description

End conditional assembly

## Syntax

```
if <expression>
instructions
endif
```

## Function

The **endif** directive closes an **if**, or **else** conditional directive. The conditional status reverts to the one existing before entering the **if** directive. The **endif** directive applies to the closest previous **if** or **else** directive.

## Example

```
if      offset != 1      ; if offset too large
addptr  offset          ; call a macro
else
        ; otherwise
inc      x               ; increment X register
endif
```

## Note

The **endif** and **endc** directives are equivalent and may be used without distinction. They are provided for compatibility with previous assemblers.

## See Also

*if, else, clist*

# endm

## Description

End macro definition

## Syntax

```
label: macro
    <macro_body>
endm
```

## Function

The **endm** directive is used to terminate macro definitions.

## Example

```
; define a macro that places the length of
; a string in a byte prior to the string
```

```
ltext:          macro
                ds.b    \@2 - \@1
\@1:
                ds.b    \1
\@2:
                endm
```

## See Also

*mexit, macro*

# endr

## Description

End repeat section

## Syntax

```
repeat  
<repeat_body>  
endr
```

## Function

The *endr* directive is used to terminate repeat sections.

## Example

```
; shift a value n times  
asln:  macro  
        repeat \1  
        sla  
        endr  
        endm  
  
; use of above macro  
asln 10          ;shift 10 times
```

## See Also

*repeat, repeatl, rexit*



# equ

## Description

Give a permanent value to a symbol

## Syntax

```
label: equ <expression>
```

## Function

The **equ** directive is used to associate a permanent value to a symbol (label). Symbols declared with the **equ** directive may not subsequently have their value altered otherwise the **set** directive should be used. *<expression>* must be either a constant expression, or a relocatable expression involving a symbol declared in the same section as the current one.

The **equ** directive can also be used to define a **bit** symbol by suffixing the defining expression with an absolute bit number. The expression and the bit number are separated by a colon character ':'. The expression can be absolute or relocatable.

## Example

```
false: equ 0 ; initialize these values
true: equ 1
tablen: equ tabfin - tabsta ;compute table length
nul: equ $0 ;define strings for ascii characters
soh: equ $1
stx: equ $2
etx: equ $3
eot: equ $4
enq: equ $5

PORTB: equ $1
PB2: equ PORTB:2
```

## See Also

*lit, set*

# even

## Description

Assemble next byte at the next even address relative to the start of a section.

## Syntax

```
even [fill_<value>]
```

## Function

The **even** directive forces the next assembled byte to the next even address. If a byte is added to the section, it is set to the value of the filling byte defined by the **-f** option. If **<fill\_value>** is specified, it will be used locally as the filling byte, instead of the one specified by the **-f** option.

## Example

```
vowtab: dc.b    'aeiou'
        even    ; ensure aligned at even address
tentab: dc.w    1, 10, 100, 1000
```

# fail

## Description

Generate error message.

## Syntax

```
fail "string"
```

## Function

The **fail** directive outputs “*string*” as an error message. No output file is produced as this directive creates an assembly error. *fail* is generally used with conditional directives.

## Example

```
Max:    equ      512
        ifge     value - Max
        fail     "Value too large"
```

# if

## Description

Conditional assembly

## Syntax

<b>if</b> <expression>	<b>or</b>	<b>if</b> <expression>
instructions		instructions
<b>endif</b>		<b>else</b>
		instructions
		<b>endif</b>

## Function

The **if**, **else** and **endif** directives allow conditional assembly. The **if** directive is followed by a constant expression. If the result of the expression is **not** zero, the following instructions are assembled up to the next matching **endif** or **else** directive; otherwise, the following instructions up to the next matching **endif** or **else** directive are skipped.

If the **if** statement ends with an **else** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endif**. So, if the **if** expression was **not** zero, the instructions between **else** and **endif** are skipped; otherwise, the instructions between **else** and **endif** are assembled. An **else** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

## Example

```
if      offset != 1      ; if offset too large
addptr  offset          ; call a macro
else    ; otherwise
inc     x               ; increment X register
endif
```

## See Also

*else, endif, clist*

# ifc

## Description

Conditional assembly

## Syntax

<b>ifc</b> <string1>,<string2>	<b>orifc</b> <string1>,<string2>
instructions	instructions
<b>endc</b>	<b>elsec</b>
	instructions
	<b>endc</b>

## Function

The **ifc**, **elsec** and **endc** directives allow conditional assembly. The **ifc** directive is followed by a constant expression. If <string1> and <string2> are equals, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped.

If the **ifc** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endc**. So, if the **ifc** expression was **not** zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

## Example

```

ifc      "hello", \2      ; if "hello" equals argument
ld       a,#45            ; load 45
elsec    ; otherwise...
ld       a,#0
endc

```

## See Also

*elsec, endc, clist*

# ifdef

## Description

Conditional assembly

## Syntax

<b>ifdef</b> <label>	<b>or</b>	<b>ifdef</b> <label>
instructions		instructions
<b>endc</b>		<b>elsec</b>
		instructions
		<b>endc</b>

## Function

The **ifdef**, **elsec** and **endc** directives allow conditional assembly. The **ifdef** directive is followed by a label <label>. If <label> is defined, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped. <label> must be first defined. It cannot be a forward reference.

If the **ifdef** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endif**. So, if the **ifdef** expression was **not** zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

## Example

```
ifdef    offset1           ; if offset1 is defined
addptr  offset1           ; call a macro
elsec                    ; otherwise
addptr  offset2           ; call a macro
endif
```

## See Also

*ifndef, elsec, endc, clist*

# ifeq

## Description

Conditional assembly

## Syntax

<b>ifeq</b> <expression> instructions endc	<b>or</b>	<b>ifeq</b> <expression> instructions elsec instructions endc
--	-----------	---

## Function

The **ifeq**, **elsec** and **endc** directives allow conditional assembly. The **ifeq** directive is followed by a constant expression. If the result of the expression is **equal** to zero, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped.

If the **ifeq** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endc**. So, if the **ifeq** expression is **equal** to zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

## Example

```
ifeq      offset      ; if offset nul
tnz       a            ; just test it
elsec     ; otherwise
add       a,#offset    ; add to accu
endc
```

## See Also

*elsec, endc, clist*

# ifge

## Description

Conditional assembly

## Syntax

<b>ifge</b> <expression> instructions endc	<b>or</b>	<b>ifge</b> <expression> instructions elsec instructions endc
--	-----------	---

## Function

The **ifge**, **elsec** and **endc** directives allow conditional assembly. The **ifge** directive is followed by a constant expression. If the result of the expression is **greater or equal** to zero, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped.

If the **ifge** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endc**. So, if the **ifge** expression is **greater or equal** to zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

## Example

```
ifge      offset-127      ; if offset too large
addptr    offset          ; call a macro
elsec     ; otherwise
inc       x               ; increment X register
endc
```

## See Also

*elsec, endc, clist*



# ifgt

## Description

Conditional assembly

## Syntax

<b>ifgt</b> <expression> instructions endc	<b>or</b>	<b>ifgt</b> <expression> instructions elsec instructions endc
--	-----------	---

## Function

The **ifgt**, **elsec** and **endc** directives allow conditional assembly. The **ifgt** directive is followed by a constant expression. If the result of the expression is **greater than** zero, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped.

If the **ifgt** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endc**. So, if the **ifgt** expression was **greater** than zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

## Example

```

ifgt      offset-127      ; if offset too large
addptr    offset          ; call a macro
elsec     ; otherwise
inc       x               ; increment X register
endc

```

## See Also

*elsec, endc, clist*

# ifl

## Description

Conditional assembly

## Syntax

<b>ifl</b> <expression> instructions endc	<b>or</b>	<b>ifl</b> <expression> instructions elsec instructions endc
---	-----------	--

## Function

The **ifl**, **elsec** and **endc** directives allow conditional assembly. The **ifl** directive is followed by a constant expression. If the result of the expression is **less or equal** to zero, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped.

If the **ifl** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endc**. So, if the **ifl** expression was **less or equal** to zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

## Example

```
ifl      offset-127      ; if offset small enough
inc      x                ; increment X register
elsec    ; otherwise
addptr   offset          ; call a macro
endc
```

## See Also

*elsec, endc, clist*

# iflt

## Description

Conditional assembly

## Syntax

<b>iflt</b> <expression> instructions endc	<b>or</b>	<b>iflt</b> <expression> instructions elsec instructions endc
--	-----------	---

## Function

The **iflt**, **else** and **endc** directives allow conditional assembly. The **iflt** directive is followed by a constant expression. If the result of the expression is **less than** zero, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped.

If the **iflt** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endc**. So, if the **iflt** expression was **less than** zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

## Example

```

iflt      offset-127      ; if offset small enough
inc       x               ; increment X register
elsec                      ; otherwise
addptr    offset         ; call a macro
endc

```

## See Also

*elsec, endc, clist*

# ifndef

## Description

Conditional assembly

## Syntax

<b>ifndef</b> <label>	<b>or</b>	<b>ifndef</b> <label>
instructions		instructions
<b>endc</b>		<b>elsec</b>
		instructions
		<b>endc</b>

## Function

The **ifndef**, **else** and **endc** directives allow conditional assembly. The **ifndef** directive is followed by a label <label>. If <label> is not defined, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped. <label> must be first defined. It cannot be a forward reference.

If the **ifndef** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endif**. So, if the **ifndef** expression was **not** zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

## Example

```
ifndef    offset1           ; if offset1 is not defined
addptr    offset2           ; call a macro
elsec     ; otherwise
addptr    offset1           ; call a macro
endif
```

## See Also

*ifdef, elsec, endc, clist*

# ifne

## Description

Conditional assembly

## Syntax

<b>ifne</b> <expression> instructions endc	<b>or</b>	<b>ifne</b> <expression> instructions elsec instructions endc
--	-----------	---

## Function

The **ifne**, **elsec** and **endc** directives allow conditional assembly. The **ifne** directive is followed by a constant expression. If the result of the expression is **not equal** to zero, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped.

If the **ifne** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endc**. So, if the **ifne** expression was **not equal** to zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

## Example

```

ifne      offset          ; if offset not nul
add       a,#offset       ; add to accu
elsec     ; otherwise
tnz       a               ; just test it
endc

```

## See Also

*elsec, endc, clist*

# ifnc

## Description

Conditional assembly

## Syntax

<b>ifnc</b> <string1>,string2> or <b>ifnc</b> <string1><string2>	
instructions	instructions
<b>endc</b>	<b>elsec</b>
	instructions
	<b>endc</b>

## Function

The **ifnc**, **elsec** and **endc** directives allow conditional assembly. The **ifnc** directive is followed by a constant expression. If <string1> and <string2> are different, the following instructions are assembled up to the next matching **endc** or **elsec** directive; otherwise, the following instructions up to the next matching **endc** or **elsec** directive are skipped.

If the **ifnc** statement ends with an **elsec** directive, the expression result is inverted and the same process applies to the following instructions up to the next matching **endc**. So, if the **ifnc** expression was **not** zero, the instructions between **elsec** and **endc** are skipped; otherwise, the instructions between **elsec** and **endc** are assembled. An **elsec** directive applies to the closest previous **if** directive.

The **if** directives may be nested. The skipped lines may or may not be in the listing depending on the **clist** directive status.

## Example

```
ifnc      "hello", \2
addptr   offset          ; call a macro
else      ; otherwise
inc       x              ; increment X register
endif
```

## See Also

*elsec, endc, clist*

# include

## Description

Include text from another text file

## Syntax

```
include "filename"
```

## Function

The **include** directive causes the assembler to switch its input to the specified *filename* until end of file is reached, at which point the assembler resumes input from the line following the **include** directive in the current file. The directive is followed by a string which gives the name of the file to be included. This string must match exactly the name and extension of the file to be included; the host system convention for uppercase/lowercase characters should be respected.

## Example

```
include "datstr" ; use data structure library
include "bldstd" ; use current build standard
include "matmac" ; use maths macros
include "ports82" ; use ports definition
```

# list

## Description

Turn on listing during assembly.

## Syntax

<b>list</b>
-------------

## Function

The **list** directive controls the parts of the program which will be written to the listing file. It is effective if and only if listings are requested; it is ignored otherwise.

## Example

```
list      ; expand source code until end or nolist
dc.b     1,2,4,8,16
end
```

## See Also

*nolist*



# lit

## Description

Give a text equivalent to a symbol

## Syntax

```
label: lit "string"
```

## Function

The **lit** directive is used to associate a text string to a symbol (label). This symbol is replaced by the string content when parsed in any assembler instruction or directive.

## Example

```
nbr:    lit    "#5"
        ld     a,nbr           ; expand as 'ld a,#5'
```

## See Also

*equ, set*

# local

## Description

Create a new local block

## Syntax

**local**

## Function

The **local** directive is used to create a new local block. When the *local* directive is used, all temporary labels defined before the local directive will be undefined after the local label. New local labels can then be defined in the new local block. Local labels can only be referenced within their own local block. A local label block is the area between two standard labels or local directives or a combination of the two.

## Example

```
var:    ds.b    1
var2:   ds.b    1
function1:
10$:    ld      a,var
        jreq    10$
        ld      var2,a

local
10$:    ld      a,var2
        jreq    10$
        ld      var,a
        ret
```

# macro

## Description

Define a macro

## Syntax

```
label: macro
    <macro_body>
endm
```

## Function

The **macro** directive is used to define a macro. The name may be any previously unused name, a name already used as a macro, or an instruction mnemonic for the microprocessor.

Macros are expanded when the name of a previously defined macro is encountered. Operands, where given, follow the name and are separated from each other by commas.

The *<argument\_list>* is optional and, if specified, is declaring each argument by name. Each argument name is prefixed by a `\` character, and separated from any other name by a comma. An argument name is an identifier which may contain `.` and `_` characters.

The *<macro\_body>* consists of a sequence of instructions not including the directives **macro** or **endm**. It may contain macro variables which will be replaced, when the macro is expanded, by the corresponding operands following the macro invocation. These macro variables take the form `\1` to `\9` to denote the first to ninth operand respectively and `\A` to `\Z` to denote the tenth to 35th operand respectively, if the macro has been defined without any *<argument\_list>*. Otherwise, macro variables are denoted by their name prefixed by a `\` character. The macro variable name can also be enclosed by parenthesis to avoid unwanted concatenation with the remaining text. In addition, the macro variable `\#` contains the number of actual operands for a macro invocation.

The special parameter `\*` is expanded to the full list of passed arguments separated by commas.

The special parameter `\0` corresponds to an extension `<ext>` which may follow the macro name, separated by the period character `'.'`. For more information, see “[Macro Instructions](#)” on page 187.

A macro expansion may be terminated early by using the **mexit** directive which, when encountered, acts as if the end of the macro has been reached.

The sequence `'\@'` may be inserted in a label in order to allow a unique name expansion. The sequence `'\@'` will be replaced by a unique number.

A macro can not be defined within another macro.

### Example

```
; define a macro that places the length of a string
; in a byte in front of the string using numbered syntax
;
ltext: macro
    dc.b    \@2-\@1

\@1:
    dc.b    \1 ; text given as first operand
\@2:
    endm

; define a macro that places the length of a string
; in a byte in front of the string using named syntax
;
ltext: macro    \string
    dc.b    \@2-\@1

\@1:
    dc.b    \string ; text given as first operand
\@2:
    endm
```

### See Also

*endm, mexit*

# **messg**

## **Description**

Send a message out to STDOUT

## **Syntax**

```
messg "<text>"  
messg '<text>'
```

## **Function**

The **messg** directive is used to send a message out to the host system's standard output (STDOUT).

## **Example**

```
messg "Test code for debug"  
    ld      a, _#2  
    ld      _SCR, a
```

## **See Also**

*title*

# mexit

## Description

Terminate a macro definition

## Syntax

```
mexit
```

## Function

The **mexit** directive is used to exit from a macro definition before the **endm** directive is reached. *mexit* is usually placed after a conditional assembly directive.

## Example

```
ctrace:macro
    if tflag == 0
        mexit
    endif
    jp      \1
endm
```

## See Also

*endm, macro*

# **mlist**

## **Description**

Turn on or off listing of macro expansion.

## **Syntax**

```
mlist [on|off]
```

## **Function**

The **mlist** directive controls the parts of the program which will be written to the listing file produced by a macro expansion. It is effective if and only if listings are requested; it is ignored otherwise.

The parts of the program to be listed are the lines which are assembled in a macro expansion.

## **See Also**

*macro*

# nolist

## Description

Turn off listing.

## Syntax

```
nolist
```

## Function

The **nolist** directive controls the parts of the program which will be **not** written to the listing file until an **end** or a **list** directive is encountered. It is effective if and only if listings are requested; it is ignored otherwise.

## See Also

*list*

## Note

For compatibility with previous assemblers, the directive **nol** is alias to **nolist**.



# **nopage**

## **Description**

Disable pagination in the listing file

## **Syntax**

**nopage**

## **Function**

The **nopage** directive stops the pagination mechanism in the listing output. It is ignored if no listing has been required.

## **Example**

```
xref      mult, div
nopage
ds.b      charin, charout
ds.w      a, b, sum
```

## **See Also**

*plen, title*

# offset

## Description

Creates absolute symbols

## Syntax

**offset** <expression>

## Function

The *offset* directive starts an absolute section which will only be used to define symbols, and not to produce any code or data. This section starts at the address specified by <expression>, and remains active while no directive or instructions producing code or data is entered. This absolute section is then destroyed and the current section is restored to the one which was active when the *offset* directive has been entered. All the labels defined in this section become absolute symbols.

<expression> must be a valid absolute expression. It must not contain any forward or external references.

## Example

```
        offset    0
next:
        ds.b      2
buffer:
        ds.b      80

        switch    .text
size:
        ld        x,next    ; ends the offset section
```

# org

## Description

Sets the location counter to an offset from the beginning of a section.

## Syntax

```
org <expression>
```

## Function

*<expression>* must be a valid absolute expression. It must not contain any forward or external references.

For an absolute section, the first *org* before any code or data defines the starting address.

An *org* directive cannot define an address smaller than the location counter of the current section.

Any gap created by an *org* directive is filled with the byte defined by the **-f** option.

# page

## Description

Start a new page in the listing file

## Syntax

<b>page</b>
-------------

## Function

The **page** directive causes a formfeed to be inserted in the listing output if pagination is enabled by either a **title** directive or the **-ft** option.

## Example

```
xref      mult, div
page
ds.b      charin, charout
ds.w      a, b, sum
```

## See Also

*plen, title*

# plen

## Description

Specify the number of lines per pages in the listing file

## Syntax

```
plen <page_length>
```

## Function

The **plen** directive causes *<page\_length>* lines to be output per page in the listing output if pagination is enabled by either a **title** directive or the **-ft** option. If the number of lines already output on the current page is less than *<page\_length>*, then the new page length becomes effective with *<page\_length>*. If the number of lines already output on the current page is greater than or equal to *<page\_length>*, a new page will be started and the new page length is set to *<page\_length>*.

## Example

```
plen      58
```

## See Also

*page, title*

# repeat

## Description

Repeat a list of lines a number of times

## Syntax

```
repeat <expression>
    repeat_body
endr
```

## Function

The repeat directive is used to cause the assembler to repeat the following list of source line up to the next endr directive. The number of times the source lines will be repeated is specified by the expression operand. The repeat directive is equivalent to a macro definition followed by the same number of calls on that macro.

## Example

```
; shift a value n times
asln:  macro
        repeat \1
        sla      (x)
        endr
        endm

; use of above macro
asln 5
```

## See Also

*endr, repeatl, rexit*

# repeatl

## Description

Repeat a list of lines a number of times

## Syntax

```
repeatl <arguments>
    repeat_body
endr
```

## Function

The **repeatl** directive is used to cause the assembler to repeat the following list of source line up to the next **endr** directive. The number of times the source lines will be repeated is specified by the number of arguments, separated with commas (with a maximum of 36 arguments) and executed each time with the value of an argument. The **repeatl** directive is equivalent to a macro definition followed by the same number of calls on that macro with each time a different argument. The repeat argument is denoted **\1** unless the argument list is starting by a name prefixed by a **\** character. In such a case, the repeat argument is specified by its name prefixed by a **\** character.

A **repeatl** directive may be terminated early by using the **rexit** directive which, when encountered, acts as if the end of the **repeatl** has been reached.

## Example

```
; test a value using the numbered syntax
repeatl      1,2,3
    add      a,#\1          ; add to accu
endr
end

or

; test a value using the named syntax
repeatl      \count,1,2,3
    add      a,#\count      ; add to accu
endr
end
```

will both produce:

```
2          ; test a value
9 0000 ab01 add    a,#1      ; add to accu
9 0002 ab02 add    a,#2      ; add to accu
9 0004 ab03 add    a,#3      ; add to accu
10         end
```

## See Also

*endr, repeat, rexit*



# restore

## Description

Restore saved section

## Syntax

```
restore
```

## Function

The **restore** directive is used to restore the last saved section. This is equivalent to a switch to the saved section.

## Example

```
                switch    .bss
var:    ds.b    1
var2:   ds.b    1
        save
        switch    .text

function1:
10$:    ld      a,var
        jreq    10$
        ld      var2,a

function2:
10$:    ld      a,var2
        sub     a,var
        jrne    10$
        ret
        restore
var3:   ds.b    1
var4:   ds.b    1

        switch    .text

        ld      a,var3
        ld      var4,a

end
```

## See Also

*save, section*

# rexit

## Description

Terminate a repeat definition

## Syntax

**rexit**

## Function

The **rexit** directive is used to exit from a **repeat** definition before the **endr** directive is reached. *rexit* is usually placed after a conditional assembly directive.

## Example

```
; shift a value n times
asln:  macro
        repeat \1
        if \1 == 0
                rexit
        endif
        sla
        endr
        endm

; use of above macro
asln 5
```

## See Also

*endr; repeat, repeatl*

# save

## Description

Save section

## Syntax

<b>save</b>
-------------

## Function

The **save** directive is used to save the current section so it may be restored later in the source file.

## Example

```
                switch    .bss
var:    ds.b    1
var2:   ds.b    1
        save
        switch    .text

function1:
10$:    ld        a,var
        jreq      10$
        ld        var2,a

function2:
10$:    ld        a,var2
        sub       a,var
        jrne      10$
        ret
        restore
var3:   ds.b    1
var4:   ds.b    1

        switch    .text

        ld        a,var3
        ld        var4,a

end
```

## See Also

*restore, section*

# scross

## Description

Turn on or off section crossing

## Syntax

```
scross [on|off]
```

## Function

The **scross** directive controls the branch instructions optimization and forces the usage of **jpf** instruction if **scross** is set (**on**) or **jp** instruction otherwise. The assembler starts with **scross** on by default.

# section

## Description

Define a new section

## Syntax

```
<section_name>: section [<attributes>]
```

## Function

The **section** directive defines a new section, and indicates that the following program is to be assembled into a section named *<section\_name>*. The *section* directive cannot be used to redefine an already existing section. If no name and no attributes are specified to the section, the default is to defined the section as a *text* section with its same attributes. It is possible to associate *<attributes>* to the new section. An attribute is either the name of an existing section or an attribute keyword. Attributes may be added if prefixed by a '+' character or not prefixed, or deleted if prefixed by a '-' character. Several attributes may be specified separated by commas. Attribute keywords are:

<b>abs</b>	absolute section
<b>bss</b>	bss style section (no data)
<b>hilo</b>	values are stored in descending order of significance
<b>even</b>	enforce even starting address and size
<b>zpage</b>	enforce 8 bit relocation
<b>long</b>	enforce 32 bit relocation
<b>bit</b>	bit section

## Example

```
CODE:  section  .text    ; section of text
lab1:  ds.b     5
DATA:  section  .data    ; section of data
lab2:  ds.b     6
      switch   CODE
lab3:  ds.b     7
      switch   DATA
lab4:  ds.b     8
```

This will place **lab1** and then **lab3** into consecutive locations in section CODE and **lab2** and **lab4** in consecutive locations in section DATA.

```
.frame:          section          .bsct,even
```

The *.frame* section is declared with same attributes than the *.bsct* section and with the *even* attribute.

```
.bit:           section          +zpage,+even,-hilo
```

The *.bit* section is declared using 8 bit relocation, with an even alignment and storing data with an ascending order of significance.

When the **-m** option is used, the *section* directive also accepts a number as operand. In that case, a labelled directive is considered as a section definition, and an unlabelled directive is considered as a section opening (*switch*).

```
.rom:           section 1          ; define section 1
               nop
.ram:           section 2          ; define section 2
               dc.b    1
               section 1          ; switch back to section 1
               nop
```

It is still possible to add attributes after the section number of a section definition line, separated by a comma.

## See Also

*switch, bsct*

# set

## Description

Give a resetable value to a symbol

## Syntax

```
label: set <expression>
```

## Function

The **set** directive allows a value to be associated with a symbol. Symbols declared with **set** may be altered by a subsequent **set**. The **equ** directive should be used for symbols that will have a constant value. *<expression>* must be fully defined at the time the **equ** directive is assembled.

## Example

```
OFST:  set      10
```

## See Also

*equ, lit*

# spc

## Description

Insert a number of blank lines before the next statement in the listing file.

## Syntax

```
spc <num_lines>
```

## Function

The **spc** directive causes <num\_lines> blank lines to be inserted in the listing output before the next statement.

## Example

```
spc      5
title    "new file"
```

If listing is requested, 5 blank lines will be inserted, then the title will be output.

## See Also

*title*



# switch

## Description

Place code into a section.

## Syntax

```
switch <section_name>
```

## Function

The **switch** directive switches output to the section defined with the **section** directive. *<section\_name>* is the name of the target section, and has to be already defined. All code and data following the *switch* directive up to the next *section*, *switch*, *bsct* or *end* directive are placed in the section *<section\_name>*.

## Example

```
                switch    .bss
buffer:         ds.b      512
                xdef      buffer
```

This will place **buffer** into the *.bss* section.

## See Also

*section*, *bsct*

# tabs

## Description

Specify the number of spaces for a tab character in the listing file

## Syntax

```
tabs <tab_size>
```

## Function

The **tabs** directive sets the number of spaces to be substituted to the tab character in the listing output. The minimum value of *<tab\_size>* is 0 and the maximum value is 128.

## Example

```
tabs      6
```

# title

## Description

Define default header

## Syntax

```
title "name"
```

## Function

The **title** directive is used to enable the listing pagination and set the default page header used when a new page is written to the listing output.

## Example

```
title    "My Application"
```

## See Also

*page, plen*

## Note

For compatibility with previous assemblers, the directive **ttl** is alias to **title**.

# xbit

## Description

Declare bit symbol as being defined elsewhere

## Syntax

```
xbit[.b] identifier[,identifier...]
```

## Function

Visibility of bit symbols between modules is controlled by the **xref** and **xbit** directives. Symbols which are defined in other modules must be declared as *xbit*. A symbol may be declared both *xdef* and *xbit* in the same module, to allow for usage of common headers.

The directive *xbit.b* declares external symbols located in the *.bsct* section.

## Example

```
xbit      otherprog
xbit.b    zpage          ; is in .bsct section
```

## See Also

*xdef*, *xref*

# xdef

## Description

Declare a variable to be visible

## Syntax

```
xdef identifier[,identifier...]
```

## Function

Visibility of symbols between modules is controlled by the **xdef** and **xref** directives. A symbol may only be declared as *xdef* in one module. A symbol may be declared both *xdef* and *xref* in the same module, to allow for usage of common headers.

## Example

```
        xdef      sqrt      ; allow sqrt to be called
                                ; from another module
sqrt:                                ; routine to return a square root
                                ; of a number >= zero
```

## See Also

*xbit*, *xref*

# xref

## Description

Declare symbol as being defined elsewhere

## Syntax

```
xref[.b] identifier[,identifier...]
```

## Function

Visibility of symbols between modules is controlled by the **xref** and **xdef** directives. Symbols which are defined in other modules must be declared as *xref*. A symbol may be declared both *xdef* and *xref* in the same module, to allow for usage of common headers.

The directive *xref.b* declares external symbols located in the *.bsct* section.

## Example

```
xref      otherprog  
xref.b    zpage          ; is in .bsct section
```

## See Also

*xbit, xdef*

# Using The Linker

---

This chapter discusses the **clnk** linker and details how it operates. It describes each linker option, and explains how to use the linker's many special features. It also provides example linker command lines that show you how to perform some useful operations. This chapter includes the following sections:

- Introduction
- Overview
- Linker Command File Processing
- Linker Options
- Section Relocation
- Setting Bias and Offset
- Linking Objects
- Linking Library Objects
- Automatic Data Initialization
- Moveable Code

- Manual Segment Initialization
- Checksum Computation
- DEFs and REFs
- Special Topics
- Description of The Map File
- Linker Command Line Examples



## Introduction

The linker combines relocatable object files, selectively loading from libraries of such files made with *clib*, to create an executable image for standalone execution or for input to other binary reformatters.

*clnk* will also allow the object image that it creates to have local symbol regions, so the same library can be loaded multiple times for different segments, and so that more control is provided over which symbols are exposed. On microcontroller architectures this feature is useful if your executable image must be loaded into several noncontiguous areas in memory.

### NOTE

*The terms “segment” and “section” refer to different entities and are carefully kept distinct throughout this chapter. A “section” is a contiguous subcomponent of an object module that the linker treats as indivisible.*

The assembler creates several sections in each object module. The linker combines input sections in various ways, but will not break one up. The linker then maps these combined input sections into output segments in the executable image using the options you specify.

A “*segment*” is a logically unified block of memory in the executable image. An example is the code segment which contains the executable instructions.

For most applications, the “*sections*” in an object module that the linker accepts as input are equivalent to the “segments” of the executable image that the linker generates as output.

## Overview

You use the linker to build your executable program from a variety of modules. These modules can be the output of the C cross compiler, or can be generated from handwritten assembly language code. Some modules can be linked unconditionally, while others can be selected only as needed from function libraries. All input to the linker, regardless of its source, must be reduced to object modules, which are then combined to produce the program file.

The linker can be used to build freestanding programs such as system bootstraps and embedded applications. It can also be used to make object modules that are loaded one place in memory but are designed to execute somewhere else. For example, a data segment in ROM to be copied into RAM at program startup can be linked to run at its actual target memory location. Pointers will be initialized and address references will be in place.

As a side effect of producing files that can be reprocessed, *clnk* retains information in the final program file that can be quite useful. The symbol table, or list of external identifiers, is handy when debugging programs, and the utility *cobj* can be made to produce a readable list of symbols from an object file. Finally, each object module has in its header useful information such as segment sizes.

In most cases, the final program file created by *clnk* is structurally identical to the object module input to *clnk*. The only difference is that the executable file is complete and contains everything that it needs to run. There are a variety of utilities which will take the executable file and convert it to a form required for execution in specific microcontroller environments. The linker itself can perform some conversions, if all that is required is for certain portions of the executable file to be stripped off and for segments to be relocated in a particular way. You can therefore create executable programs using the linker that can be passed directly to a PROM programmer.

The linker works as follows:

- Options applying to the linker configuration. These options are referred to in this chapter as “[\*Global Command Line Options\*](#)” on page 265.
- Command file options apply only to specific sections of the object being built. These options are referred to in this chapter as “[\*Segment Control Options\*](#)” on page 267.
- Sections can be relocated to execute at arbitrary places in physical memory, or “stacked” on suitable storage boundaries one after the other.
- The final output of the linker is a header, followed by all the segments and the symbol table. There may also be an additional debug symbol table, which contains information used for debugging purposes.

## Linker Command File Processing

The command file of the linker is a small control language designed to give the user a great deal of power in directing the actions of the linker. The basic structure of the command file is a series of command items. A command item is either an explicit linker option or the name of an input file (which serves as an implicit directive to link in that file or, if it is a library, scan it and link in any required modules of the library).

An explicit linker option consists of an option keyword followed by any parameters that the option may require. The options fall into five groups:

Group 1
( <b>+seg</b> <section>) controls the creation of new segments and has parameters which are selected from the set of local flags.
( <b>+grp</b> <section>) controls the section grouping.
Group 2
( <b>+inc</b> *) is used to include files
Group 3
( <b>+new</b> , <b>+pub</b> and <b>+pri</b> ) controls name regions and takes no parameters.
Group 4
( <b>+def</b> <symbol>) is used to define symbols and aliases and takes one required parameter, a string of the form <b>ident1=ident2</b> , a string of the form <b>ident1=constant</b> , or a string of the form <b>ident1=@segment</b> .
Group 5
( <b>+spc</b> <segment>) is used to reserve space in a particular <segment> and has a required parameter

A description of each of these command line options appears below.

The manner in which the linker relocates the various sections is controlled by the **+seg** option and its parameters. If the size of a current segment is zero when a command to start a new segment of the same name is encountered, it is discarded. Several different sections can be redirected directly to the same segment by using the **+grp** option.

*clnk* links the *<files>* you specify in order. If a file is a library, it is scanned as long as there are modules to load. Only those library modules that define public symbols for which there are currently outstanding unsatisfied references are included.

## Inserting comments in Linker commands

Each input line may be ended by a comment, which must be prefixed by a **#** character. If you have to use the **#** as a significant character, you can escape it, using the syntax **\#**.

Here is an example for an indirect link file:

```
# Link for EPROM
+seg .text -b0x8100 -n .text # start eeprom address
+seg .const -a .text      # constants follow program
+seg .bsct -b 0x0 -m 0x100 # short range start address
+seg .data -b 0x100 -n .data # uninitialized data
+seg .bss -a .data -n .bss  # initialized data to 0
\cxstm8\lib\crt.s.sm8      # startup object file
mod1.o mod2.o              # input object files
\cxstm8\lib\libisl.sm8     # C library
\cxstm8\lib\libm.sm8       # machine library
+seg .vector -b0x8000 -0x7f # vectors eeprom address
vector.o                   # reset and interrupt vectors
```

## Linker Options

The linker accepts the following options, each of which is described in detail below.

```
clnk [options] <file.lkf> [<files>]
    -bs#  bank size
    -e*   error file name
    -ge   eclipse error messages
    -l*>  library path
    -m*   map file name
    -o*   output file name
    -p    phys addr in map
    -s    symbol table only
    -sa   sort symbol by address
    -si   suppress .info. segment
    -sl   output local symbols
    -u#   display unused symbols
    -v    verbose
```

The **output file name** and the **link command file** **must** be present on the command line. The options are described in terms of the two groups listed above; the global options that apply to the linker, and the segment control options that apply only to specific segments.

## Global Command Line Options

The global command line options that the linker accepts are:

### Global linker Options

Option	Description
<b>-bs#</b>	set the window shift to #, which implies that the number of bytes in a window is <b>2**#</b> . The default value is . For more information, see the section “ <a href="#">Address Specification</a> ” on page 277.
<b>-e*</b>	log errors in the text file * instead of displaying the messages on the terminal screen.
<b>-ge</b>	produce error messages directly compatible with the Eclipse environment.
<b>-l*&gt;</b>	specify library path. You can specify up to 128 different paths. Each path is a directory name, <b>not</b> terminated by any directory separator character.
<b>-m*</b>	produce map information for the program being built to file *.
<b>-o*</b>	write output to the file *. This option is required and has no default value.
<b>-p</b>	display symbols with physical address instead of logical address in the map file.
<b>-s</b>	create an output file containing only an absolute symbol table, but still with an object file format. The resulting file can then be used in another link to provide the symbol table of an existing application.
<b>-sa</b>	display symbols sort by address instead of alphabetic order in the map file.
<b>-si</b>	suppress the <b>.info</b> segment content for compatibility with tools not supporting this segment yet.
<b>-sl</b>	output local symbols in the executable file.

**Global linker Options**

Option	Description								
<b>-u#</b>	display unused symbols. Valid options are: <table><tr><td><b>-u1</b></td><td>display data symbols</td></tr><tr><td><b>-u2</b></td><td>display code and constant symbols</td></tr><tr><td><b>-u4</b></td><td>display absolute symbols (located variables)</td></tr><tr><td><b>-u8</b></td><td>display symbols defined in the link file</td></tr></table>	<b>-u1</b>	display data symbols	<b>-u2</b>	display code and constant symbols	<b>-u4</b>	display absolute symbols (located variables)	<b>-u8</b>	display symbols defined in the link file
<b>-u1</b>	display data symbols								
<b>-u2</b>	display code and constant symbols								
<b>-u4</b>	display absolute symbols (located variables)								
<b>-u8</b>	display symbols defined in the link file								
Those values can be combined (added or or'ed) to display several categories. This option has no effect when the debug option has been set as symbols are at least referenced in the debug section. Symbols defined in removed sections are of course not displayed.									
<b>-v</b>	be "verbose".								



## Segment Control Options

This section describes the segment control options that control the structure of individual segments of the output module.

A group of options to control a specific segment must begin with a **+seg** option. Such an option must precede any group of options so that the linker can determine which segment the options that follow apply to. The linker allows up to **255** different segments.

**+seg <section> <options>** start a new segment loading assembler section type *<section>* and build it as directed by the *<options>* that follow:

### Segment Control Options Usage

Option	Description
<b>-a*</b>	make the current segment follow the segment <i>*</i> , where <i>*</i> refers to a segment name given explicitly by a <b>-n</b> option. Options <b>-b</b> and <b>-e</b> cannot be specified if <b>-a</b> has been specified. Option <b>-o</b> can be specified only with value <b>0</b> to reset the logical address to the same value than the physical address.
<b>-b*</b>	set the physical start address of the segment to <i>*</i> . Option <b>-e</b> or <b>-a</b> cannot be specified if <b>-b</b> has been specified.
<b>-c</b>	do not output any code/data for the segment.
<b>-ck</b>	mark the segment you want to check. For more information, see " <a href="#">Checksum Computation</a> " on page 287.
<b>-ds#</b>	set the bank size for paged addresses calculation. This option overwrites the global <b>-bs</b> option for that segment.
<b>-e*</b>	set the physical end address of the segment to <i>*</i> . Option <b>-b</b> or <b>-a</b> cannot be specified if <b>-e</b> has been specified.
<b>-f#</b>	fill the segment up to the value specified by the <b>-m</b> option with single bytes or two byte words whose value is <i>#</i> . This option has no effect if no <b>-m</b> option is specified for that segment.

**Segment Control Options Usage (cont.)**

Option	Description										
<b>-i?</b>	define the initialization option. Valid options are: <table border="1"> <tr> <td><b>-it</b></td><td>use this segment to host the descriptor and images copies of initialized data used for automatic data initialization</td></tr> <tr> <td><b>-id</b></td><td>initialize this segment</td></tr> <tr> <td><b>-ib</b></td><td>do not initialize this segment</td></tr> <tr> <td><b>-ik</b></td><td>mark this segment as checksum segment</td></tr> <tr> <td><b>-ic</b></td><td>mark this segment as moveable segment</td></tr> </table>	<b>-it</b>	use this segment to host the descriptor and images copies of initialized data used for automatic data initialization	<b>-id</b>	initialize this segment	<b>-ib</b>	do not initialize this segment	<b>-ik</b>	mark this segment as checksum segment	<b>-ic</b>	mark this segment as moveable segment
<b>-it</b>	use this segment to host the descriptor and images copies of initialized data used for automatic data initialization										
<b>-id</b>	initialize this segment										
<b>-ib</b>	do not initialize this segment										
<b>-ik</b>	mark this segment as checksum segment										
<b>-ic</b>	mark this segment as moveable segment										
<b>-k</b>	mark the segment as a root segment for the unused section suppression. This flag is usually applied on the reset and interrupt vectors section, and as soon as it is specified at least once in the linker command file, enables the section suppression mechanism. This option can be used on any other segment to force the linker to keep it even if it is not used.										
<b>-m*</b>	set the maximum size of the segment to * bytes. If not specified, there is no checking on any segment size. If a segment is declared with the <b>-a</b> option as following a segment which is marked with the <b>-m</b> option, then set the maximum available space for all the possible consecutive segments. If a <b>-m</b> is specified on a <b>-a</b> segment, the actual maximum size checked is equal to the given value minus the size of all the segments already allocated from the first segment of the <b>-a</b> list. So the new maximum size is computed from the start address of the list and not from the start address of that segment.										

## Segment Control Options Usage (cont.)

Option	Description
<b>-n*</b>	<p>set the output name of the segment to *. Segment output names have at most <b>15</b> characters; longer names are truncated. If no name is given with a <b>-n</b> option, the segment inherits a default name equal to its assembler section name.</p> <p>For example, use this option when you want to generate the hex records for a particular PROM, such as:</p> <pre>+seg .text -b0x2000 -n prom1 &lt;object_files&gt; +seg .text -b0x4000 -n prom2 &lt;object_files&gt; ...</pre> <p>You can generate the hex records for <b>prom1</b> by typing:</p> <pre>chex -n prom1 file.sm8</pre> <p>For more information, see “<a href="#">The chex Utility</a>” in <b>Chapter 8</b>.</p>
<b>-o*</b>	<p>set the logical start address of the segment to * if <b>-b</b> option is specified or the logical end address if <b>-e</b> option is specified. The default is to set the logical address equal to the physical address. Options <b>-b</b> and <b>-e</b> cannot be specified both if <b>-o</b> has been specified.</p>
<b>-q*</b>	<p>enable a manual initialization mechanism by specifying a name of an empty segment which will host the initialization image of the current segment. See “<a href="#">Manual Segment Initialization</a>” on page 286.</p>
<b>-r*</b>	<p>round up the starting address of the segment and all the loaded sections. The expression defines the power of two of the alignment value. The option <b>-r3</b> will align the start address to an 8 bytes boundary. This option has no effect if the start address is explicitly defined by a <b>-b</b> option and has no effect on the first object module.</p>

## Segment Control Options Usage (cont.)

Option	Description
<b>-s*</b>	define a space name for the segment. This segment will be verified for overlapping only against segments defined with the same space name. See “ <a href="#">Overlapping Control</a> ” on page 277.
<b>-v</b>	do not verify overlapping for the segment.
<b>-w*</b>	set the window size for banked applications, and activate the automatic bank segment creation.
<b>-x</b>	expandable segment. Allow a segment to spill in the next segment of the same section type if its size exceeds the value given by the <b>-m</b> option. The next segment must be declared <b>before</b> the object causing the overflow. This option has no effect if no <b>-m</b> option is specified for the expandable segment. Option <b>-e</b> cannot be specified with option <b>-x</b> .

Options defining a numerical value (addresses and sizes) can be entered as constant, symbols, or simple expression combined them with ‘+’ and ‘-’ operators. Any symbol used has to be defined before to be used, either by a **+def** directive or loaded as an absolute symbol from a previously loaded object file. The operators are applied from left to right without any priority and parenthesis **()** are not allowed. Such expressions CANNOT contain any whitespace. For example:

```
+def START=0x1000
+def MAXSIZE=0x2000
+seg .text -bSTART+0x100 -mMAXSIZE-0x100
```

The first line defines the symbol `START` equals to the absolute value 1000 (hex value), the second line defines the symbol `MAXSIZE` equals to the absolute value 2000 (hex value). The last line opens a `.text` segment located at 1100 (hex value) with a maximum size of 1f00 (hex value). For more information, see the section “[Symbol Definition Option](#)” on page 274.

Unless **-b\*** is given to set the *bss* segment start address, the *bss* segment will be made to follow the last *data* segment in the output file. Unless **-b\*** is given to set the *data* segment start address, the *data* segment will be made to follow the last *bsct* segment in the output file. The *bsct* and *text* segments are set to start at zero unless you specify otherwise by using **-b** option. It is permissible for all segments to overlap, as far as *clnk* is concerned; the target machine may or may not make sense of this situation (as with separate instruction and data spaces).

### NOTE

*A new segment of the specified type will not actually be created if the last segment of the same name has a size of zero. However, the new options will be processed and will override the previous values.*

## Segment Grouping

Different sections can be redirected directly to the same segment with the **+grp** directive:

**+grp <section>=<section list>** where *<section>* is the name of the target section, and *<section list>* a list of section names separated by commas. When loading an object file, each section listed in the right part of the declaration will be loaded as if it was named as defined in the left part of the declaration. The target section may be a new section name or the name of an existing section (including the pre-defined ones). When using a new name, this directive has to be preceded by a matching **+seg** definition.

### NOTE

*Whitespaces are **not** allowed aside the equal sign '=' and the commas.*

## Linking Files on the Command line

The linker supports linking objects from the command line. The link command file has to be modified to indicate where the objects are to be loaded using the following **@#** syntax.

---

<b>@1, @2,...</b>	include each individual object file at its positional location on the command line and insert them at the respective locations in the link file ( <b>@1</b> is the first object file, and so on).
<b>@*</b>	include all of the objects on the command line and insert them at this location in the link file.

---

## Example

Linking objects from the command line:

```
clnk -o test.sm8 test.lkf file1.o file2.o
```

```
## Test.lkf:
+seg .text -b0x5000
+seg .data -b0x100
@1
+seg .text -b0x7000
@2
```

Is equivalent to

```
clnk -o test.sm8 test.lkf
## test.lkf
+seg .text -b0x5000
+seg .data -b0x100
file1.o
+seg .text -b0x7000
file2.o
```

## Include Option

Subparts of the link command file can be included from other files by using the following option:

---

<b>+inc*</b>	include the file specified by *. This is equivalent to expanding the text file into the link file directly at the location of the <b>+inc</b> line. The linker searches the specified file using the provided list of directories, unless specifying an absolute path.
--------------	--

---

## Example

Include the file “seg2.txt” in the link file “test.lkf”:

```
## Test.lkf:
+seg .text -b0x5000
+seg .data -b0x100
file1.o file2.o
+seg .text -b0x7000
+inc seg2.txt

## seg2.txt:
mod1.o mod2.o mod3.o

## Resultant link file
+seg .text -b0x5000
+seg .data -b0x100
file1.o file2.o
+seg .text -b0x7000
mod1.o mod2.o mod3.o
```

## Private Region Options

Options that control code regions are:

---

<b>+new</b>	start a new region. A “region” is a user definable group of input object modules which may have both public and private portions. The private portions of a region are local to that region and may not access or be accessed by anything outside the region. By default, a new region is given public access.
<b>+pub</b>	make the following portion of a given region public.
<b>+pri</b>	make the following portion of a given region private.

---

## Symbol Definition Option

The option controlling symbol definition and aliases is:

**+def\***      define new symbols to the linker. The string \* can use different syntaxes:

<b>ident=constant</b>	where <i>ident</i> is a valid identifier and <i>constant</i> is a valid constant expressed with the standard C language syntax. This form is used to add <i>ident</i> to the symbol table as a defined absolute symbol with a value equal to <i>constant</i> .
<b>ident=constant:bitnum</b>	where <i>ident</i> is a valid identifier, <i>constant</i> is a valid constant expressed with the standard C language syntax and <i>bitnum</i> a constant expression between 0 and 7. This form is used to add <i>ident</i> to the symbol table as a defined absolute bit symbol with a value equal to <i>constant</i> 3-bit left shifted and or'ed with <i>bitnum</i> .
<b>ident1=ident2</b>	where <i>ident1</i> and <i>ident2</i> are both valid identifiers. This form is used to define aliases. The symbol <i>ident1</i> is defined as the alias for the symbol <i>ident2</i> and goes in the symbol table as an external DEF (a DEF is an entity defined by a given module.) If <i>ident2</i> is not already in the symbol table, it is placed there as a REF (a REF is an entity referred to by a given module).
<b>ident1=ident2:bitnum</b>	where <i>ident1</i> and <i>ident2</i> are both valid identifiers, and <i>bitnum</i> a constant between 0 and 7. This form is used to define bit aliases. The symbol <i>ident1</i> is defined as the alias for the corresponding bit of symbol <i>ident2</i> which cannot be already a bit symbol itself, and goes in the symbol table as an external DEF (a DEF is an entity defined by a given module.) If <i>ident2</i> is not already in the symbol table, it is placed there as a REF (a REF is an entity referred to by a given module).
<b>ident=@section</b>	where <i>ident</i> is a valid identifier, and <i>section</i> is the name of a section specified as the first argument of a <b>+seg</b> directive. This form is used to add <i>ident</i> to the symbol table as a defined symbol whose value is the address of the next byte to be loaded in the specified section.



## NOTE

Whitespaces are **not** allowed aside the equal sign '='.

The following list of possible syntaxes uses a common construct. The segment name must match the name of the segment specified by the **-n** option. These directives can be placed anywhere in the link command file, even before the corresponding segment is defined.

<b>ident=start(segment)</b>	This form defines <i>ident</i> as the <i>logical start</i> address of the designated segment.
<b>ident=end(segment)</b>	This form defines <i>ident</i> as the <i>logical end</i> address of the designated segment.
<b>ident=pstart(segment)</b>	This form defines <i>ident</i> as the <i>physical start</i> address of the designated segment.
<b>ident=pend(segment)</b>	This form defines <i>ident</i> as the <i>physical end</i> address of the designated segment.
<b>ident=size(segment)</b>	This form defines <i>ident</i> as the <i>size</i> of the designated segment.
<b>ident=init(segment)</b>	This form defines <i>ident</i> as the <i>initialization image start</i> address of the designated segment.

## NOTE

Whitespaces are **not** allowed aside the equal sign '='.

For more information about DEFs and REFs, refer to the section “[DEFs and REFs](#)” on page 289.

## Reserve Space Option

The following option is used to reserve space in a given segment:

<b>+spc &lt;segment&gt;=&lt;value&gt;</b>	reserve <value> bytes of space at the current location in the segment named <segment>.
---	--

---

<b>+spc &lt;segment&gt;=@section</b>	reserve a space at the current location in the segment named <i>&lt;segment&gt;</i> equal to the current size of the opened segment where the given <i>section</i> is loaded. The size is evaluated at once, so if the reference segment grows after that directive, there is no further modification of the space reservation. If such a directive is used to duplicate an existing section, it has to be placed in the link command file after all the object files.
--------------------------------------	--

---

---

**NOTE**

---

Whitespaces are **not** allowed aside the equal sign '='.

---

## Handle Dependencies

This directive allows creating or suppressing a dependency between two functions using their assembly level symbol:

---

<b>+dep &lt;func1&gt;+&lt;func2&gt;</b>	add a dependence marking <i>&lt;func1&gt;</i> as calling <i>&lt;func2&gt;</i> .
<b>+dep &lt;func1&gt;-&lt;func2&gt;</b>	suppress a dependence marking <i>&lt;func1&gt;</i> as not calling <i>&lt;func2&gt;</i> .

---

This directive is mostly used to help building complex applications using a static model.

---

**NOTE**

---

Whitespaces are **not** allowed aside the + and - signs.

---

## Section Relocation

The linker relocates the sections of the input files into the segments of the output file.

An absolute section, by definition, cannot and should not be relocated. The linker will detect any conflicts between the placement of this file and its absolute address given at compile/assemble time.

In the case of a bank switched system, it is still possible for an absolute section to specify a physical address different from the one and at compile/assembly time, the logical address **MUST** match the one specified at compile/assemble time.

### Address Specification

The two most important parameters describing a segment are its **bias** and its **offset**, respectively its physical and logical start addresses. In nonsegmented architectures there is no distinction between *bias* and *offset*. The *bias* is the address of the location in memory where the segment is relocated to run. The *offset* of a segment will be equal to the *bias*. In this case you must set only the *bias*. The linker sets the *offset* automatically.

### Overlapping Control

The linker is verifying that a segment does not overlap any other one, by checking the physical addresses (*bias*). This control can be locally disabled for one segment by using the **-v** option. For targets implementing separated address spaces (such as bank switching), the linker allows several segments to be isolated from the other ones, by giving them a *space* name with the **-s** option. In such a case, a segment in a named space is checked only against the other segments of the same space. The unnamed segments are checked together.

## Setting Bias and Offset

The bias and offset of a segment are controlled by the **-b\*** option and **-o\*** option. The rules for dealing with these options are described below.

## Setting the Bias

If the **-b\*** option is specified, the bias is set to the value specified by \*. Otherwise, the bias is set to the end of the last segment of the same name. If the **-e\*** option is specified, the bias is set to value obtain by subtracting the segment size to the value specified by \*.

## Setting the Offset

If the **-o\*** option is specified, the offset is set to the value specified by \*. Otherwise, the offset is set equal to the bias.

## Using Default Placement

If none of **-b**, **-e** or **-o** options is specified, the segment may be placed *after* another one, by using the **-a\*** option, where \* is the name of another segment. Otherwise, the linker will try to use a default placement based on the segment name. The compiler produces specific sections for code (*.text*) and data (*.data*, *.bss*, *.bsect* and *.ubsect*). By default, *.text* and *.bsect* segments start at zero, *.ubsect* segment follows the latest *.bsect* segment, *.data* segment follows the latest *.text* segment, and *.bss* segment follows the latest *.data* segment. Note that there is no default placement for the constants segment *.const* and the bit segment *.bit*.

## Bit Segment Handling

Bit segments are allocated using bit addresses. A bit address is a value based on the byte address and the bit number in this byte. The bit address is equal to the byte address 3-bit left shifted or'ed with the bit number. The bias (or offset) value can be entered directly as a bit address or with a special syntax combining the byte address and the bit number. The following lines are identical:

```
+seg .bit -b 0x103
+seg .bit -b 0x20:3
```

When using the **-a** option, the linker automatically converts byte address to bit address when entering a bit segment from a byte segment, starting at bit 0, and converts a bit address to a byte address when leaving a bit segment to a byte segment, starting from next available byte.

Bit addresses are displayed in the map file using the combined syntax.

If the bit segment contains initialized variables, the code must be linked with an appropriate startup file and the bit segment must be declared with option **-id** in the linker command file. Otherwise, the bit segment must be declared with option **-c** to suppress it from the output file.

## Linking Objects

A new segment is built by concatenating the corresponding sections of the input object modules in the order the linker encounters them. As each input section is added to the output segment, it is adjusted to be relocated relative to the end portion of the output segment so far constructed. The first input object module encountered is relocated relative to a value that can be specified to the linker. The size of the output *bss* segment is the sum of the sizes of the input *bss* sections.

Unless the **-v** option has been specified on a segment definition, the linker checks that the segment physical address range does not overlap any other segment of the application. Logical addresses are not checked as bank switching creates several segments starting at the same logical address.

## Linking Library Objects

The linker will selectively include modules from a library when outstanding references to member functions are encountered. The library file must be placed **after** all objects that may call its modules to avoid unresolved references. The standard ANSI libraries are provided in two versions to provide the level of support that your application needs. This can save a significant amount of code space and execution time when full ANSI single precision floating point support is not needed. The first letter after “*lib*” in each library file denotes the library type (**f** for single precision, and **i** for integer). See below.

---

### **libf.sm8**

Single Precision Library. This library is used for applications where only single precision floating point support is needed. Link this library **before** the other libraries when **only** single precision floats are used.

---

### **libi.sm8**

Integer only Library. This library is designed for applications where **no** floating point is used. Floats can still be used for arithmetic but not with the standard library. Link this library **before** the other libraries when only integer libraries are needed.

---

**libl(0).sm8** Eeprom Library for STM8L. This library is designed to give access to the STM8L eeprom functions. When used, this library **MUST** be linked **before** any other libraries.

Memory Model	Machine Library	Integer Only Library	Float Library
Stack Short	libm(0).sm8	libis(0).sm8	libfs(0).sm8
Stack Long		libisl(0).sm8	libfsl(0).sm8

### NOTE

*When using a model for application smaller than 64K, you must link with the specific set of libraries (names ending with '0').*

## Library Order

You should link your application with the libraries in the following orders:

Integer Only Application	Single Precision Float Application
libi.sm8	libf.sm8
libm.sm8	libi.sm8
	libm.sm8

For more information, see “[Linker Command Line Examples](#)” on page 298.

## Libraries Setup Search Paths

The linker uses the environment variable **CXLIB** to search for objects and library files. If you don’t specify the full path to the objects and/or libraries in the link command file AND they are not found in the local directory, the linker will then search all paths specified by the **CXLIB** environment variable. This allows you to specify just the names of the objects and libraries in your link command file. For example, setting the **CXLIB** environment variable to the **C:\COSMIC\LIB** directory is done as follow:

```
C>set CXLIB=C:\COSMIC\LIB
```



## Automatic Data Initialization

The linker is able to configure the executable for an automatic data initialization. This mechanism is initiated automatically when the linker finds the symbol `__idesc__` in the symbol table, as an *undefined* symbol. `clnk` first locates a segment behind which it will add an image of the data, so called the *host* segment. The default behaviour is to select the **first** *.text* segment in the executable file, but you can override this by marking one segment with the **-it** option.

Then, `clnk` looks in the executable file for initialized segments. All the segments *.data* and *.bss* are selected by default, unless disabled explicitly by the **-ib** option. Otherwise, renamed segments may also be selected by using the **-id** option. The **-id** option cannot be specified on a bss segment, default or renamed. Once all the selected segments are located, `clnk` builds a descriptor containing the starting address and length of each such segment, and moves the descriptor and the selected segments to the end of the *host* segment, without relocating the content of the selected segments.

For more information, see “[Generating Automatic Data Initialization](#)” in **Chapter 2** and “[Initializing data in RAM](#)” in **Chapter 3**.

### Descriptor Format

The created descriptor has the following format:

```
dc.w start_ram_address; starting address of the
    ; first image in prom
; for each segment:
dc.b flag                ; segment type
dc.w start_ram_address  ; start address of segment in ram
dc.w end_prom_address   ; address of last data byte
    ; plus one in prom
; after the last segment:
dc.b 0
```

The flag byte is used to detect the end of the descriptor, and also to specify a type for the data segment. The actual value is equal to the code of the first significant letter in the segment name.

If the RAM segment has been created using banked addresses (**-b** and **-o** values), the RAM start address is described using two words, the first

giving the page value for that segment and the second giving the matching value for the start address in that space. A segment description is displayed as:

```
dc.b flag                ; segment type
dc.w start_ram_address   ; start address of segment in ram
dc.w end_prom_address    ; address of last data byte
```

The end address in PROM of one segment gives also the starting address in prom of the following segment, if any.

The address of the descriptor will be assigned to the symbol `__idesc__`, which is used by the *crti.s* startup routine. So all this mechanism will be activated just by linking the *crti.sm8* file with the application, or by referencing the symbol `__idesc__` in your own startup file.

If the *host* segment has been opened with a **-m** option giving a maximum size, *clnk* will check that there is enough space to move all the selected segments.

## Moveable Code

The linker allows a code segment to be stored in the ROM part, but linked at another address which is supposed to be located in RAM. This feature is specially designed to allow an application to run FLASH programming routines or bootloader from the RAM space. This feature is sharing the same global mechanism than initialized data, and the common descriptor built by the linker contains both record types. The flag byte is used to qualify each entry. In order to implement such a feature, the link command file should contain a dedicated code segment marked with the **-ic** option:

```
#          LINKER EXAMPLE FOR MOVEABLE CODE
#
# mark this segment with -ic and link it at RAM address
#
+seg .text -b 0x100 -n boot -ic
flash.o
+seg .text -b 0x8000 -n code# application code
file.o
...
```

The function contained in the object `flash.o` is now linked at the RAM address `0x100` but stored somewhere in the code space along with any other initialized data. It is not necessary to link the application with the startup routine `crt0.s` if the application does not contain initialized data but the descriptor will be built as soon as a moveable function is used by the application, but if the `crt0.s` startup is used, moveable code segments are **not** copied in RAM at the application start up.

In order to use such a function, it is necessary to first copy it from ROM to RAM. This is done by calling the library function `_fctcpy()` with one character argument equal to the first significant letter of the moveable segment name. This argument allows an application to implement several different moveable segments for different kind of situations. In such a case, all the moveable segment names should have names with different first character. This function returns a boolean status equal to 0 if no moveable segment has been copied, or a value different of zero otherwise. Once the segment has been successfully copied, the RAM function can be called directly:

```
if (_fctcpy('b'))  
    flash();
```

There is no possible name conflict between data segment names and moveable code segment names because the linker internally marks the flag byte differently.

## Manual Segment Initialization

The linker allows a segment to be manually initialized at runtime by calling a copy routine such as *memcpy()*. For such a segment, the linker builds a data image linked at the initial segment address but stored at a different location. The target address is usually in RAM while the source address is usually in flash. In order to enable this mechanism, an empty named segment (**-n\***) must be created somewhere in the flash space and the initialized segment must be created with a **-q\*** option specifying the name of the flash segment containing the image. The start addresses of both segments and their length will be obtained with the usage of appropriate **+def** directives.

Assuming we have a function created in a section named **.ramcode**, the following linker command file abstract shows how to define the appropriate segments:

```
+seg .text -b 0x8000 -n code
+seg .image -a code -n image
+seg .ramcode -b 0x3000 -n ramcode -q image
```

The code built and expected to run at 0x3000 will be stored in the **.image** segment wherever it is located, and will not be loaded at its execution location in the resulting application code. It is then possible to use *memcpy()* when the code must be copied to its execution location (0x3000 in this case). You can define the following linker symbols in your linker command file for the execution start address (copy to), the image location address (copy from) and the image size.

```
+def _ramcode=start(ramcode)
+def _image=start(image)
+def _isize=size(image)
```

The linker symbol assignments are treated like addresses so the copy code would be the following:

```
extern char ramcode, image, isize;

memcpy(&ramcode, &image, (unsigned int)&isize);
```

## Checksum Computation

This feature is activated by the detection of the symbol `__ckdesc__` as an undefined symbol. This is practically done by calling one of the provided checksum functions, which uses that symbol and returns `0` if the checksum is correct. These functions are provided in the integer library and are the following:

<code>__checksum()</code>	check a 8 bit checksum stored once for all the selected segments.
<code>__checksumx()</code>	check a 8 bit checksum stored for every selected segments. This method allows a segment to be dynamically reloaded by updating the corresponding CRC byte.
<code>__checksum16()</code>	check a 16 bit checksum stored once for all the selected segments.
<code>__checksum16x()</code>	check a 16 bit checksum stored for every selected segments. This method allows a segment to be dynamically reloaded by updating the corresponding CRC word.

You then have to update the link command file in two ways:

- 1) Mark the segments (usually code segments) you want to check, by using the `-ck` option on the `+seg` line. Note that you need only to mark the first segment of a hooked list, meaning that if a segment is declared with `-a` option as following a segment which is marked with the `-ck` option, it will automatically inherit the `-ck` marker and will be also checked. Note also that if you are using the automatic initialization mechanism, and if the code segment hosting the init descriptor (`-it`) is also marked with `-ck`, the init segment and ALL the initialization copy segments will also be checked.
- 2) Create an empty segment which will contain the checksum descriptor. This has to be an empty segment, located wherever you want with a `-b` or `-a` option. This segment will NOT be checked, even if marked or hooked to a marked segment. The linker will fill this segment with a data descriptor allowing the checking function to scan

all the requested segments and compute the final **crc**. This segment has to be specially marked with the option **-ik** to allow the linker to recognize it as the checksum segment.

Here is an example of link command file showing how to use **-ck** and **-ik**:

```
#          LINKER EXAMPLE FOR CHECKSUM IMPLEMENTATION
#
# mark the first segment of an attached list with -ck
#
+seg .text -b 0x8000 -n code -ck# this segment is marked
+seg .const -a code -n const# this one is implicitly marked
#
# create an empty segment for checksum table marked with -ik
#
+seg .cksum -a const -n cksum -ik # checksum segment
#
# remaining part should contain the verification code
#
+seg .data -b 0x100
crtsex.sm8
test.o
libi.sm8
libm.sm8
+def __memory=@.bss
```

The descriptor built by the linker is a list of entries followed by the expected CRC value, only once if functions `_checksum()` or `_checksum16()` are called, or after each entry if functions `_checksumx()` or `_checksum16x()` are called. An entry contains a flag byte, a start address and an end address. The flag byte is non-zero, and is *or'ed* with **0x80** if the start address contains a bank value (two words, page first then start address), otherwise it is just one word with the start address. The end address is always one word. The last entry is always followed by a nul byte (seen as an ending flag), and immediately followed by the expected CRC if functions `_checksum()` or `_checksum16()` are called. The linker compresses the list of entries by creating only one entry for contiguous segments (as long as they are in the same space (**-s\*** option) and in the same bank/page).

The current linker implements only one algorithm. Starting with zero, the CRC byte/word is first rotated one bit left (a true bit rotation), then xor'ed with the code byte. The CRC values stored in the checksum descriptor are the one's complement value of the expected CRC.

## DEFs and REFs

The linker builds a new symbol table based on the symbol tables in the input object modules, but it is not a simple concatenation with adjustments. There are two basic type of symbols that the linker puts into its internal symbol table: **REFs** and **DEFs**. DEFs are symbols that are defined in the object module in which they occur. REFs are symbols that are referenced by the object module in which they occur, but are not defined there.

The linker also builds a debug symbol table based on the debug symbol tables in any of the input object modules. It builds the debug symbol table by concatenating the debug symbol tables of each input object module in the order it encounters them. If debugging is not enabled for any of input object module, the debug symbol table will be of zero length.

An incoming REF is added to the symbol table as a REF if that symbol is not already entered in the symbol table; otherwise, it is ignored (that reference has already been satisfied by a DEF or the reference has already been noted). An incoming DEF is added to the symbol table as a DEF if that symbol is not already entered in the symbol table; its value is adjusted to reflect how the linker is relocating the input object module in which it occurred. If it is present as a REF, the entry is changed to a DEF and the symbol's adjusted value is entered in the symbol table entry. If it is present as a DEF, an error occurs (multiply defined symbol).

When the linker is processing a library, an object module in the library becomes an input object module to the linker only if it has at least one DEF which satisfies some outstanding REF in the linker's internal symbol table. Thus, the simplest use of *clnk* is to combine two files and check that no unused references remain.

The executable file created by the linker must have no REFs in its symbol table. Otherwise, the linker emits the error message “*undefined symbol*” and returns failure.

## Special Topics

This section explains some special linker capabilities that may have limited applicability for building most kinds of microcontroller applications.

### Private Name Regions

*Private name regions* are used when you wish to link together a group of files and expose only some to the symbol names that they define. This lets you link a larger program in groups without worrying about names intended only for local usage in one group colliding with identical names intended to be local to another group. Private name regions let you keep names truly local, so the problem of name space pollution is much more manageable.

An explicit use for private name regions in a STM8 environment is in building a paged program with duplication of the most used library functions in each page, in order to avoid extra page commutation. To avoid complaints when multiple copies of the same file redefine symbols, each such contribution is placed in a private name region accessible only to other files in the same page.

The basic sequence of commands for each island looks like:

```
+new <public files> +pri <private libraries>
```

Any symbols defined in *<public files>* are known outside this private name region. Any symbols defined in *<private libraries>* are known only within this region; hence they may safely be redefined as private to other regions as well.

### NOTE

*All symbols defined in a private region are local symbols and will not appear in the symbol table of the output file.*



## Renaming Symbols

At times it may be desirable to provide a symbol with an alias and to hide the original name (*i.e.*, to prevent its definition from being used by the linker as a DEF which satisfies REFs to that symbol name). As an example, suppose that the function *func* in the C library provided with the compiler does not do everything that is desired of it for some special application. There are three methods of handling this situation (we will ignore the alternative of trying to live with the existing function's deficiencies).

The first method is to write a new version of the function that performs as required and link it into the program being built before linking in the libraries. This will cause the new definition of *func* to satisfy any references to that function, so the linker does not include the version from the library because it is not needed. This method has two major drawbacks: first, a new function must be written and debugged to provide something which basically already exists; second, the details of exactly what the function must do and how it must do it may not be available, thus preventing a proper implementation of the function.

The second approach is to write a new function, say *my\_func*, which does the extra processing required and then calls the standard function *func*. This approach will generally work, unless the original function *func* is called by other functions in the libraries. In that case, the extra function behavior cannot occur when *func* is called from library functions, since it is actually *my\_func* that performs it.

The third approach is to use the aliasing capabilities of the linker. Like the second method, a new function will be written which performs the new behavior and then calls the old function. The twist is to give the old function a new name and hide its old name. Then the new function is given the old function's name and, when it calls the old function, it uses the new name, or alias, for that function. The following linker script provides a specific example of this technique for the function *func*:

```
line 1 +seg .text -b 0x1000
line 2 +seg .data -b0
line 3 +new
line 4 Crts.xx
line 5 +def _oldfunc=_func
```

```
line 6 +pri func.o
line 7 +new
line 8 prog.o newfunc.o
line 9 <libraries>
```

### NOTE

*The function name `func` as referenced here is the name as seen by the C programmer. The name which is used in the linker for purposes of aliasing is the name as seen at the object module level. For more information on this transformation, see the section “[Interfacing C to Assembly Language](#)” in **Chapter 3**.*

The main thing to note here is that `func.o` and `new_func.o` both define a (different) function named `func`. The second function `func` defined in `newfunc.o` calls the old `func` function by its alias `oldfunc`.

Name regions provide limited scope control for symbol names. The **+new** command starts a new name region, which will be in effect until the next **+new** command. Within a region there are public and private name spaces. These are entered by the **+pub** and **+pri** commands; by default, **+new** starts in the public name space.

**Lines 1,2** are the basic linker commands for setting up a separate I/D program. Note that there may be other options required here, either by the system itself or by the user.

**Line 3** starts a new region, initially in the public name space.

**Line 4** specifies the startup code for the system being used.

**Line 5** establishes the symbol `_oldfunc` as an alias for the symbol `_func`. The symbol `_oldfunc` is entered in the symbol table as a public definition. The symbol `_func` is entered as a private reference in the current region.

**Line 6** switches to the private name space in the current region. Then `func.o` is linked and provides a definition (private, of course) which satisfies the reference to `_func`.

**Line 7** starts a new name region, which is in the public name space by default. Now no reference to the symbol `_func` can reach the definition created on **Line 6**. That definition can only be reached now by using the symbol `_oldfunc`, which is publicly defined as an alias for it.

**Line 8** links the user program and the module `newfunc.o`, which provides a new (and public) definition of `_func`. In this module the old version is accessed by its alias. This new version will satisfy all references to `_func` made in `prog.o` and the libraries.

**Line 9** links in the required libraries.

The rules governing which name space a symbol belongs to are as follows:

- Any symbol definition in the public space is public and satisfies all outstanding and future references to that symbol.
- Any symbol definition in the private space of the current region is private and will satisfy any private reference in the current region.
- All private definitions of a symbol must occur before a public definition of that symbol. After a public definition of a symbol, any other definition of that symbol will cause a “*multiply defined symbol*” error.
- Any number of private definitions are allowed, but each must be in a separate region to prevent a multiply defined symbol error.
- Any new reference is associated with the region in which the reference is made. It can be satisfied by a private definition in that region, or by a public definition. A previous definition of that symbol will satisfy the reference if that definition is public, or if the definition is private and the reference is made in the same region as the definition.
- If a new reference to a symbol occurs, and that symbol still has an outstanding unsatisfied reference made in another region, then that symbol is marked as requiring a public definition to satisfy it.

- Any definition of a symbol must satisfy all outstanding references to that symbol; therefore, a private definition of a symbol which requires a public definition causes a blocked symbol reference error.
- No symbol reference can “reach” any definition made earlier than the most recent definition.

## Absolute Symbol Tables

*Absolute Symbol tables* are used to export symbols from one application to another, to share common functions for instance, or to use functions already built in a ROM, from an application downloaded into RAM. The linker option **-s** will modify the output file in order to contain only a symbol table, without any code, but still with an object file format, by using the same command file used to build the application itself. All symbols are flagged as *absolute* symbols. This file can be used in another link, and will then transmit its symbol table, allowing another application to use those symbols as *externals*. Note that the linker does not produce any map even if requested, when used with the **-s** option.

The basic sequence of commands looks like:

```
clnk -o appli.sm8 -m appli.map appli.lkf  
clnk -o appli.sym -s appli.lkf
```

The first link builds the application itself using the *appli.lkf* command file. The second link uses the same command file and creates an object file containing only an absolute symbol table. This file can then be used as an input object file in any other link command file.

## Description of The Map File

The linker can output a map file by using the **-m** option. The map file contains 4 sections: the *Segment* section, the *Modules* section, the *Stack Usage* section and the *Symbols* section.

**Segment** Describe the different segments which compose the application, specifying for each of them: the start address (in hexa), the end address (in hexa), the length (in decimal), and the name of the segment. Note that the end value is the address of the byte following the last one of the segment, meaning that an empty segment will have the same start and end addresses. If a segment is initialized, it is displayed twice, the first time with its final address, the second time with the address of the image copy.

**Modules** List all the modules which compose the application, giving for each the description of all the defined sections with the same format as in the *Segment* section. If an object has been assembled with the **-pl** option, local symbols are displayed just after the module description.

**Stack Usage** Describe the amount of memory needed for the stack. Each function of the application is listed by its name, followed by a ‘>’ character indicating that this function is not called by any other one (the *main* function, *interrupt* functions, *task* entries...). The first number is the total size of the stack used by the function including all the internal calls. The second number between braces shows the stack need for that function alone. The entry may be flagged by the keyword “**Recursive**” meaning that this function is itself recursive or is calling directly or indirectly a recursive function, and that the total stack space displayed is not accurate. The linker may detect potential but not actual recursive functions when such functions are called by pointer. The linker displays at the end of the list a total stack size assuming interrupt functions cannot be themselves interrupted. Interrupt frames and machine library calls are properly counted.

**Call Tree** List all the functions sorted alphabetically followed by all the functions called inside. The display goes on recursively unless a function has already been listed. In such a case, the name is followed by the line number where the function is expanded. If a line becomes too long, the process is suspended and the line ends with a ... sequence indicating that this function is listed later. Functions called by pointer are listed between parenthesis, or between square brackets if called from an array of pointers.

**Symbols** List all the symbols defined in the application specifying for each its name, its value, the section where it is defined, and the modules where it is used. If the target processor supports bank switching, addresses are displayed as logical addresses by default. Physical addresses can be displayed by specifying the **-p** option on the linker command line. Addresses of bit symbols are displayed with the byte address followed by a colon character and the bit number.

### Special Segments **.debug** and **.info**.

The map file also displays two informational segments that are not defined in the link command file and users should not create or attempt to relocate them. Both segments should always show a starting address of at 0x0. These segments are produced by the compiler and are part of the linked executable, but they do NOT generate any code or use any target resources. All information is processed and used on the host.

**.debug segment** The information in the **.debug** segment is used to provide function information such as local variables and source line information for use with a source level debugger. This segment is created when the **+debug** option is used when compiling or **-xx** option on the assembler. The information is read directly by Cosmic's ZAP debuggers or converted to other formats such as ELF/DWARF or IEEE-695 using Cosmic supplied utilities *cvdwarf* and *cv695*. The **.debug** segment information may also be extracted in text format using the Cosmic *cprd* utility included with the compiler.

**.info. segment** The **.info.** segment is generated automatically by the compiler and it contains the component version and options used to compile and link the application. This information can be extracted into a text file using the *cobj* utility with the **-i** option and **-o** options. This segment may also be suppressed by the linker using the linker command line option **-si**. This option is useful for compatibility with older debuggers and utilities.

## Return Value

*clnk* returns success if no error messages are printed to STDOUT; that is, if no undefined symbols remain and if all reads and writes succeed. Otherwise it returns failure.

## Linker Command Line Examples

This section shows you how to use the linker to perform some basic operations.

A linker command file consists of linker options, input and output file, and libraries. The options and files are read from a command file by the linker. For example, to create an STM8 file from *file.o* you can type at the system prompt:

```
clnk -o myapp.sm8 myapp.lkf
```

where *myapp.lkf* contains:

```
+seg .text -b0xf000 -n .text # start eprom address
+seg .const -a .text        # constants follow program
+seg .bsct -b0x0 -niram -m 0x100# initialized short range
+seg .data -b0x100          # start data address
\cxstm8\lib\crt.s.sm8       # startup object file
file1.o file2.o            # input object files
\cxstm8\lib\libis.sm8       # C library
\cxstm8\lib\libm.sm8        # machine library
+def __memory=@.bss         # symbol used by startup
```

The following link command file is an example for an application that does **not** use floating point data types and does **not** require automatic initialization.

```
# demo.lkf: link command WITHOUT automatic init
+seg .text -b 0xf000 -n.text # program start address
+seg .const -a .text        # constants follow program
+seg .bsct -b0x0 -niram -m 0x100# initialized short range
+seg .data -b0x100          # start data address
\cxstm8\lib\crt.s.sm8       # startup with NO-INIT
acia.o                     # main program
module1.o                  # module program
\cxstm8\lib\libis.sm8       # C library
\cxstm8\lib\libm.sm8        # machine library
+seg .const -b0x8000        # vectors eprom address
vector.o                   # reset & interrupt vectors
# define these symbols if crt.s is used
# +def __endzp=@.ubsct      # end of uninitialized zpage
# +def __memory=@.bss       # symbol used by library
```



The following link command file is an example for an application that uses single precision floating point data types and utilizes automatic data initialization.

```
# demo.lkf: link command WITH automatic init
+seg .text -bf000 0x -n.text # program start address
+seg .const -a .text        # constants follow program
+seg .bsct -b0x80 -niram -m 0x80# initialized short range
+seg .ubsct -n iram          # uninitialized short range
+seg .data -b0x100           # start data address
\cxstm8\lib\crtsi.sm8      # startup with auto-init
acia.o                      # main program
module1.o                   # module program
\cxstm8\lib\libfs.sm8       # single precision library
\cxstm8\lib\libis.sm8       # integer library
\cxstm8\lib\libm.sm8        # machine library
+seg .const -b0x8000        # vectors eeprom address
vector.o                    # reset & interrupt vectors
# define these symbols if crtsi is used
+def __endzp=@.ubsct        # end of uninitialized zpage
+def __memory=@.bss         # end of bss segment
```



# Debugging Support

---

This chapter describes the debugging support available with the cross compiler targeting the [STM8](#). There are two levels of debugging support available, so you can use either the COSMIC's **Zap** C source level cross debugger or your own debugger or in-circuit emulator to debug your application. This chapter includes the following sections:

- [Generating Debugging Information](#)
- [Generating Line Number Information](#)
- [Generating Data Object Information](#)
- [The cprd Utility](#)
- [The clst utility](#)

## Generating Debugging Information

The compiler generates debugging information in response to command line options you pass to the compiler as described below. The compiler can generate the following debugging information:

- 1 line number information that allows COSMIC's C source level debugger or another debugger or emulator to locate the address of the code that a particular C source line (or set of lines) generates. You may put line number information into the object module in either of the two formats, or you can generate both line number information and information about program data and function arguments, as described below.
- 2 information about the name, type, storage class and address (absolute or relative to a stack offset) of program static data objects, function arguments, and automatic data objects that functions declare. Information about what source files produced which relocatable or executable files. This information may be localized by address (where the output file resides in memory). It may be written to a file, sorted by address or alphabetical order, or it may be output to a printer in paginated or unpaginated format.

### Generating Line Number Information

The compiler puts line number information into a special debug symbol table. The debug symbol table is part of the relocatable object file produced by a compilation. It is also part of the output of the *clnk* linker. You can therefore obtain line number information about a single file, or about all the files making up an executable program. However, the compiler can produce line number information only for files that are **fewer** than 65,535 lines in length.

### Generating Data Object Information

The **+debug** option directs the compiler to generate information about data objects and function arguments and return types. The debugging information the compiler generates is the information used by the COSMIC's C source level cross debugger or another debugger or emulator. The information produced about data objects includes their name, scope, type and address. The address can be either absolute or relative to a stack offset.

As with line number information alone, you can generate debugging information about a single file or about all the files making up an executable program.

*cprd* may be used to extract the debugging information from files compiled with the **+debug** option, as described below.

## The *cprd* Utility

**cprd** extracts information about functions and data objects from an object module or executable image that has been compiled with the **+debug** option. *cprd* extracts and prints information on the name, type, storage class and address (absolute or offset) of program static data objects, function arguments, and automatic data objects that functions declare. For automatic data, the address provided is an offset from the frame pointer. For function arguments, the address provided is an offset from the stack pointer.

### Command Line Options

*cprd* accepts the following command line options, each of which is described in detail below:

```
cprd [options] file
      -fc* select function name
      -fl* select file name
      -o*  output file name
      -ra  recurse structure always
      -r   recurse structure once
      -s   display object size
      -u   display unused object
```

where *<file>* is an object file compiled from C source with the compiler command line option **+debug** set.

### Cprd Option Usage

Option	Description
<b>-fc*</b>	print debugging information only about the function *. By default, <i>cprd</i> prints debugging information on all functions in <i>&lt;file&gt;</i> . Note that information about global data objects is always displayed when available.
<b>-fl*</b>	print debugging information only about the file *. By default, <i>cprd</i> prints debugging information on all C source files.
<b>-o*</b>	print debugging information to file *. Debugging information is written to your terminal screen by default.

**Cprd Option Usage (cont.)**

Option	Description
<b>-ra</b>	display any occurrence of structure fields with their offset.
<b>-r</b>	display the first occurrence of structure fields with their offset.
<b>-s</b>	display object size in bytes.
<b>-u</b>	display only unused global variables.

By default, *cprd* prints debugging information about all functions and global data objects in *<file>*.

**Examples**

The following example show sample output generated by running the *cprd* utility on an object file created by compiling the program *acia.c* with the compiler option **+debug** set.

```
cprd acia.sm8
```

```
Information extracted from acia.sm8
```

```
source file acia.c:
```

```
unsigned char buffer[64] at 0x0104
unsigned char *ptlec at 0x0102
unsigned char *ptecr at 0x0100
```

```
unsigned char getch() lines 26 to 36 at 0x810a-0x8135
    auto unsigned char c at -1 from frame pointer
```

```
void outch() lines 40 to 45 at 0x8136-0x8144
    argument unsigned char c at 0 from frame pointer
```

```
void receipt() lines 51 to 57 at 0x8145-0x8167
    (no locals)
```

```
void main() lines 63 to 72 at 0x8168-0x818a
    (no locals)
```

```
source file vector.c:
```

```
void (*_vectab[16])() at 0x8000
```

## The *clst* utility

The **clst** utility takes relocatable or executable files as arguments, and creates listings showing the C source files that were compiled or linked to obtain those relocatable or executable files. It is a convenient utility for finding where the source statements are implemented.

To use *clst* efficiently, its argument files must have been compiled with the **+debug** option.

*clst* can be instructed to limit its display to files occupying memory in a particular range of addresses, facilitating debugging by excluding extraneous data. *clst* will display the entire content of any files located between the endpoints of its specified address range.

### Command Line Options

*clst* accepts the following command line options, each of which is described in detail below:

```
clst [options> file
    -a  list file alphabetically
    -b  display physical address
    -f*> process selected file
    -i*> source file directory
    -l#  page length
    -o*  output file name
    -p  suppress pagination
    -r*  specify a line range #:#
```

### Clst Option Usage

Option	Description
<b>-a</b>	when set, cause <i>clst</i> to list files in alphabetical order. The default is that they are listed by increasing addresses.
<b>-b</b>	display physical address instead of logical address in the listing file.
<b>-f*&gt;</b>	specify <b>*</b> as the file to be processed. Default is to process all the files of the application. Up to 10 files can be specified.



**Clst Option Usage (cont.)**

Option	Description
<b>-i*&gt;</b>	read string <i>*</i> to locate the source file in a specific directory. Source files will first be searched for in the current directory, then in the specified directories in the order they were given to <i>clst</i> . You can specify up to 10 different paths. Each path is a directory name, <b>not</b> terminated by any directory separator character.
<b>-l#</b>	when paginating output, make the listings <i>#</i> lines long. By default, listings are paginated at 66 lines per page.
<b>-o*</b>	redirect output from <i>clst</i> to file <i>*</i> . You can achieve a similar effect by redirecting output in the command line. <div><pre>clst -o acia.lst acia.sm8</pre></div> is equivalent to: <div><pre>clst acia.sm8 &gt;acia.lst</pre></div>
<b>-p</b>	suppress pagination. No page breaks will be output.
<b>-r#:#</b>	where <i>#:#</i> is a range specification. It must be of the form <i>&lt;number&gt;:&lt;number&gt;</i> . When this flag is specified, only those source files occupying memory in the specified range will be listed. If part of a file occupies memory in the specified range, that file will be listed in its entirety. The following is a valid use of <b>-r</b> : <div><pre>-r 0xe000:0xe200</pre></div>



# CHAPTER 8

## Programming Support

This chapter describes each of the programming support utilities packaged with the C cross compiler targeting the [STM8](#). The following utilities are available:

Utility	Description
<b>chex</b>	translate object module format
<b>clabs</b>	generate absolute listings
<b>clib</b>	build and maintains libraries
<b>cobj</b>	examine objects modules
<b>ctat</b>	generate short range variables header file
<b>cvdwarf</b>	generate ELF/DWARF format

The assembler is described in **Chapter 5**, “[Using The Assembler](#)”. The linker is described in **Chapter 6**, “[Using The Linker](#)”. Support for debugging is described in **Chapter 7**, “[Debugging Support](#)”.

The description of each utility tells you what tasks it can perform, the command line options it accepts, and how you use it to perform some commonly required operations. At the end of the chapter are a series of examples that show you how to combine the programming support utilities to perform more complex operations.

## The *chex* Utility

You use the ***chex*** utility to translate executable images produced by *clnk* to one of several hexadecimal interchange formats. These formats are: *Motorola S-record* format, and *Intel standard hex* format. You can also use *chex* to override text and data biases in an executable image or to output only a portion of the executable.

The executable image is read from the input file *<file>*.

### Command Line Options

*chex* accepts the following command line options, each of which is described in detail below:

```
chex [options] file
-a##      absolute file start address
-b##      address bias
-e##      entry point address
-f?       output format
-h        suppress header
+h*       specify header string
-m#       maximum data bytes per line
-n*>      output only named segments
-o*       output file name
-p        use paged address format
-pa       use paged address for data
-pl##     page number for linear mapping
-pn       use paged address in bank only
-pp       use paged address with mapping
-s        output increasing addresses
-w        output word addresses
-x*>      exclude named segments
```

### Chex Option Usage

Option	Description
<b>-a##</b>	the argument file is considered as a pure binary file and ## is the output address of the first byte.
<b>-b##</b>	subtract ## to any address before output.

## Chex Option Usage (cont.)

Option	Description								
<b>-e##</b>	define <b>##</b> as the entry point address encoded in the dedicated record of the output format, if available.								
<b>-f?</b>	define output file format. Valid options are: <table border="1"> <tr> <td><b>i</b></td><td>Intel Hex Format</td></tr> <tr> <td><b>m</b></td><td>Motorola S19 format</td></tr> <tr> <td><b>2</b></td><td>Motorola S2 format</td></tr> <tr> <td><b>3</b></td><td>Motorola S3 format</td></tr> </table> <p>Default is to produce Motorola S-Records (<b>-fm</b>). Any other letter will select the default format</p>	<b>i</b>	Intel Hex Format	<b>m</b>	Motorola S19 format	<b>2</b>	Motorola S2 format	<b>3</b>	Motorola S3 format
<b>i</b>	Intel Hex Format								
<b>m</b>	Motorola S19 format								
<b>2</b>	Motorola S2 format								
<b>3</b>	Motorola S3 format								
<b>-h</b>	do not output the header sequence if such a sequence exists for the selected format.								
<b>+h*</b>	insert * in the header sequence if such a sequence exists for the selected format.								
<b>-m#</b>	output # maximum data bytes per line. Default is to output 32 bytes per line.								
<b>-n*&gt;</b>	output only segments whose name is equal to the string *. Up to twenty different names may be specified on the command line. If there are several segments with the same name, they will all be produced. This option is used in combination with the <b>-n</b> option of the linker.								
<b>-o*</b>	write output module to file *. The default is STDOUT.								
<b>-p</b>	output addresses of banked segments using a paged format <b>&lt;page_number&gt;&lt;logical_address&gt;</b> , instead of the default format <b>&lt;physical&gt;</b> .								
<b>-pa</b>	output addresses of banked data segments using a paged format <b>&lt;page_number&gt;&lt;logical_address&gt;</b> , instead of the default format <b>&lt;physical&gt;</b> .								

## Chex Option Usage (cont.)

Option	Description
<b>-pl##</b>	specify the page value of the segment localized between 0x8000 and 0xc000 when using a linear non-banked application. This option enforces a paged format for this segment.
<b>-pn</b>	behaves as <b>-p</b> but only when logical address is inside the banked area. This option has to be selected when producing an hex file for the Noral debugger.
<b>-pp</b>	behaves as <b>-p</b> but uses paged addresses for all banked segments, mapped or unmapped. This option has to be selected when producing an hex file for Promic tools.
<b>-s</b>	sort the output addresses in increasing order.
<b>-w</b>	output word addresses. Addresses must be aligned on even addresses. This option is useful for word processor type.
<b>-x*&gt;</b>	do not output segments whose name is equal to the string *. Up to twenty different names may be specified on the command line. If there are several segments with the same name, they will not all be output.

## Return Status

*chex* returns success if no error messages are printed; that is, if all records are valid and all reads and writes succeed. Otherwise it returns failure.

## Examples

The file *hello.c*, consisting of:

```
char *p = {"hello world"};
```

when compiled produces the following the following *Motorola S-record* format:

```
chex hello.o
```

```
S00A000068656C6C6F2E6F44
S1110000020068656C6C6F20776F726C640090
S9030000FC
```

and the following *Intel standard hex* format:

```
chex -fi hello.o  
:0E0000000020068656C6C6F20776F726C640094  
:00000001FF
```

## The clabs Utility

**clabs** processes assembler listing files with the associated executable file to produce listing with updated code and address values.

*clabs* decodes an executable file to retrieve the list of all the files which have been used to create the executable. For each of these files, *clabs* looks for a matching listing file produced by the compiler (“**.ls**” file). If such a file exists, *clabs* creates a new listing file (“**.la**” file) with absolute addresses and code, extracted from the executable file.

To be able to produce any results, the compiler **must** have been used with the ‘**-l**’ option.

### Command Line Options

*clabs* accepts the following command line options, each of which is described in detail below.

```
clabs [options] file
    -a      process also library files
    -cl*    listings files
    -l      restrict to local directory
    -p      use paged address format
    -pn     use paged address in bank only
    -pp     use paged address with mapping
    -r*     relocatable listing suffix
    -s*     absolute listing suffix
    -v      echo processed file names
```

### Clabs Option Usage

Option	Description
<b>-a</b>	process also files located in libraries. Default is to process only all the files of the application.
<b>-cl*</b>	specify a path for the listing files. By default, listings are created in the same directory than the source files.
<b>-l</b>	process files in the current directory only. Default is to process all the files of the application.



**Clabs Option Usage (cont.)**

Option	Description
<b>-p</b>	output addresses of banked segments using a paged format <code>&lt;page_number&gt;&lt;logical_address&gt;</code> , instead of the default format <code>&lt;physical&gt;</code> .
<b>-pn</b>	behaves as <b>-p</b> but only when logical address is inside the banked area.
<b>-pp</b>	behaves as <b>-p</b> but uses paged addresses for all banked segments, mapped or unmapped.
<b>-r*</b>	specify the input suffix, including or not the dot '.' character. Default is <code>".ls"</code>
<b>-s*</b>	specify the output suffix, including or not the dot '.' character. Default is <code>".la"</code>
<b>-v</b>	be verbose. The name of each module of the application is output to STDOUT.

`<file>` specifies one file, which must be in executable format.

**Return Status**

*clabs* returns success if no error messages are printed; that is, if all reads and writes succeed. An error message is output if no relocatable listing files are found. Otherwise it returns failure.

**Examples**

The following command line:

```
clabs -v acia.sm8
```

will output:

```
crts.ls
acia.ls
vector.ls
```

and creates the following files:

```
crts.la
acia.la
vector.la
```

The following command line:

```
clabs -r.lst acia.sm8
```

will look for files with the suffix “**.lst**”:

The following command line:

```
clabs -s.lx acia.sm8
```

will generate:

```
crt.s.lx  
acia.lx  
vector.lx
```

## The *clib* Utility

***clib*** builds and maintains object module libraries. *clib* can also be used to collect arbitrary files in one place. *<library>* is the name of an existing library file or, in the case of replace or create operations, the name of the library to be constructed.

### Command Line Options

*clib* accepts the following command line options, each of which is described in detail below:

```
clib [options] <library> <files>
-a      accept absolute symbols
-c      create a new library
-d      delete modules from library
-e      accept empty module
-i*     object list filename
-l      load all library at link
-p      no pathname in modules
-r      replace modules in library
-s      list symbols in library
-t      list files in library
-v      be verbose
-x      extract modules from library
```

### Clib Option Usage

Option	Description
<b>-a</b>	include absolute symbols in the library symbol table.
<b>-c</b>	create a library containing <i>&lt;files&gt;</i> . Any existing <i>&lt;library&gt;</i> of the same name is removed before the new one is created.
<b>-d</b>	delete from the library the zero or more files in <i>&lt;files&gt;</i> .
<b>-e</b>	accept module with no symbol.
<b>-i*</b>	take object files from a list *. You can put several files per line or put one file per line. Each lines can include comments. They must be prefixed by the '#' character. If the command line contains <i>&lt;files&gt;</i> , then <i>&lt;files&gt;</i> will be also added to the library.

### Clib Option Usage (cont.)

Option	Description
<b>-l</b>	when a library is built with this flag set, all the modules of the library will be loaded at link time. By default, the linker only loads modules necessary for the application.
<b>-p</b>	do not prefix module names in library with any path. This option is only meaningful with option <b>-c</b> .
<b>-r</b>	in an existing library, replace the zero or more files in <i>&lt;files&gt;</i> . If no library <i>&lt;library&gt;</i> exists, create a library containing <i>&lt;files&gt;</i> . The files in <i>&lt;files&gt;</i> not present in the library are added to it.
<b>-s</b>	list the symbols defined in the library with the module name to which they belong.
<b>-t</b>	list the files in the library.
<b>-v</b>	be verbose
<b>-x</b>	extract the files in <i>&lt;files&gt;</i> that are present in the library into discrete files with the same names. If no <i>&lt;files&gt;</i> are specified, all files in the library are extracted.

At most one of the options **-[c r t x]** may be specified at the same time. If none of these is specified, the **-t** option is assumed.

### Return Status

*clib* returns success if no problems are encountered. Otherwise it returns failure. After most failures, an error message is printed to STDERR and the library file is not modified. Output from the **-t**, **-s** options, and verbose remarks, are written to STDOUT.

### Examples

To build a library and check its contents:

```
clib -c libc one.o two.o three.o
clib -t libc
```

will output:

```
one.o  
two.o  
three.o
```

To build a library from a list file:

```
clib -ci list libc six.o seven.o
```

where *list* contains:

```
# files for the libc library  
one.o  
two.o  
three.o  
four.o  
five.o
```

## The *cobj* Utility

You use ***cobj*** to inspect relocatable object files or executable. Such files may have been output by the assembler or by the linker. *cobj* can be used to check the size and configuration of relocatable object files or to output information from their symbol tables.

### Command Line Options

*cobj* accepts the following options, described in detail below.

<b><i>cobj</i></b>	<b>[options]</b>	<b>file</b>
<b>-d</b>	output data flows	
<b>-h</b>	output header	
<b>-i</b>	display info section	
<b>-n</b>	output sections	
<b>-o*</b>	output file name	
<b>-r</b>	output relocation flows	
<b>-s</b>	output symbol table	
<b>-v</b>	display file addresses	
<b>-x</b>	output debug symbols	

*<file>* specifies a file, which must be in relocatable format or executable format.

### Cobj Option Usage

Option	Description
<b>-d</b>	output in hexadecimal the data part of each section.
<b>-h</b>	display all the fields of the object file header.
<b>-i</b>	display the content of the <b>.info</b> section in a readable format.
<b>-n</b>	display the name, size and attribute of each section.
<b>-o*</b>	write output module to file *. The default is STDOUT.
<b>-r</b>	output in symbolic form the relocation part of each section.
<b>-s</b>	display the symbol table.
<b>-v</b>	display seek addresses inside the object file.
<b>-x</b>	display the debug symbol table.

If none of these options is specified, the default is **-hns**.

## Return Status

*cobj* returns success if no diagnostics are produced (*i.e.* if all reads are successful and all file formats are valid).

## Examples

For example, to get the symbol table:

```
cobj -s acia.o
```

symbols:

```
_main:      0000003e section .text defined public
_outch:     0000001b section .text defined public
_buffer:    00000000 section .bss defined public
_ptecr:     00000000 section .bsct defined public zpage
_getch:     00000000 section .text defined public
_ptlec:     00000002 section .bsct defined public zpage
_recept:    00000028 section .text defined public
```

The information for each symbol is: name, address, section to which it belongs and attribute.

## The Ctat Utility

You use the **ctat** utility to produce a file which can be preincluded to the next compilation. The output file contains variables which can be efficiently allocated in the short range section.

### Command Line Options

**ctat** accepts the following command line options, each of which is described in detail below:

```
ctat [options] file
      -cl*   listings directory
      -f*    input header file
      -l     restrict to local directory
      -m#    maximum short range size
      -o*    output file name
      -v     verbose
```

### Ctat Option Usage

Option	Description
<b>-cl*</b>	specify a path for the listing files. By default, listings are in the same directory than the source files.
<b>-f*</b>	specify an input header file.
<b>-l</b>	specify only listings in the local directory are used.
<b>-m#</b>	specify the maximum short range size.
<b>-o*</b>	write output to the file *. The default is STDOUT.
<b>-v</b>	be "verbose".

*<file>* is an executable whose objects are compiled with command line options **-l** and **+debug**, in order to produce listings and debug information.

This utility scans the debug information to build a list of global variables not already defined in a dedicated space and the listing files to create a count usage of these variables. If a header file is entered with the option **-f** containing a list of variables described using the same syntax



than the output file, the specified variables will be ignored for the allocation.

The most used variables are then allocated in the short range section by producing a specific header file containing a list of **#pragma attribute** directives. The maximum available space in the short range section is defaulted to 256 bytes minus the space already used by the provided application. The maximum space can be modified by using the **-m** option.

The created header file can then be included by recompiling the application sources with the option **-ph** followed by the header file name.

## Return Status

*ctat* returns success if no problems are encountered. Otherwise it returns failure.

## Examples

The following command:

```
ctat -o zpage.h test.sm8
```

will generate **zpage.h** as the result file from the application *test.sm8*. The application must be recompiled by adding to the already specified options:

```
cxstm8 -phzpage.h ...
```

## The cvdwarf Utility

**cvdwarf** is the utility used to convert a file produced by the linker into an **ELF/DWARF** format file.

### Command Line Options

*cvdwarf* accepts the following options, each of which is described in detail below.

```
cvdwarf [options] file
    -bp##    bank start address
    -bs#     bank shift
    +dup     accept duplicate headers
    -e*      entry symbol or address
    -loc     complex location description
    -o*      output file name
    +page#   define pagination (HC12/HCS08 only)
    -rb     reverse bitfield (L to R)
    -so     add stack offset
    -v      be verbose
```

<file> specifies a file, which must be in executable format.

### Cvdwarf Option usage

Option	Description
<b>-bp#</b>	start address of the banking page.
<b>-bs#</b>	set the window shift to #, which implies that the number of bytes in a window is <b>2**#</b> .  <b>THESE FLAGS ARE CURRENTLY ONLY MEANINGFULL FOR THE HC11K4.</b>
<b>+dup</b>	handle duplicate header files individually. By default, the converter assumes that all header files sharing the same name do have the same content or with conditional behaviours.
<b>-e*</b>	specify either a symbol name or a numerical value, defining the entry field in the elf header. If a symbol name is specified, it is searched in the assembler symbol table. The value of symbol <b>__stext</b> is used by default.

## Cvdwarf Option usage (cont.)

Option	Description																
<b>-loc</b>	location lists are used in place of location expressions whenever the object whose location is being described can change location during its lifetime. <b>THIS POSSIBILITY IS NOT SUPPORTED BY ALL DEBUGGERS.</b>																
<b>-o*</b>	where * is a filename. * is used to specify the output file for <i>cvdwarf</i> . By default, if <b>-o</b> is not specified, <i>cvdwarf</i> send its output to the file whose name is obtained from the input file by replacing the filename extension with <b>“.elf”</b> .																
<b>+page#</b>	output addresses in paged mode where # specifies the page type:																
	<table><tr><th>#</th><th></th><th>Valid usage for</th><th>Paging Window</th></tr><tr><td>1</td><td>for banked code</td><td>All HC12, HCS12 and HCS08 paged derivatives when Code Paging used</td><td>FLASH 0x8000 to 0xbfff</td></tr><tr><td>2</td><td>for banked data</td><td>Only for HC12A4 when Data Paging used</td><td>RAM 0x7000 to 0x7fff</td></tr><tr><td>3</td><td>both (code and data)</td><td>Only for HC12A4 when Data and Code Paging used</td><td>FLASH 0x8000 to 0xbfff RAM 0x7000 to 0x7fff</td></tr></table>	#		Valid usage for	Paging Window	1	for banked code	All HC12, HCS12 and HCS08 paged derivatives when Code Paging used	FLASH 0x8000 to 0xbfff	2	for banked data	Only for HC12A4 when Data Paging used	RAM 0x7000 to 0x7fff	3	both (code and data)	Only for HC12A4 when Data and Code Paging used	FLASH 0x8000 to 0xbfff RAM 0x7000 to 0x7fff
#		Valid usage for	Paging Window														
1	for banked code	All HC12, HCS12 and HCS08 paged derivatives when Code Paging used	FLASH 0x8000 to 0xbfff														
2	for banked data	Only for HC12A4 when Data Paging used	RAM 0x7000 to 0x7fff														
3	both (code and data)	Only for HC12A4 when Data and Code Paging used	FLASH 0x8000 to 0xbfff RAM 0x7000 to 0x7fff														
	By default, the banked mode is disable.																
	<b>THIS FLAG IS CURRENTLY ONLY MEANINGFULL FOR THE HC12/HCS12 and HCS08.</b>																
	<b>THIS FLAG IS NOT TO BE USED ON ANY S12X PAGING, BASED ON THE EXISTING GLOBAL ADDRESS MODE.</b>																
<b>-rb</b>	reverse bitfield from left to right.																

By default, the banked mode is disable.

**THIS FLAG IS CURRENTLY ONLY MEANINGFULL FOR THE HC12/HCS12 and HCS08.**

**THIS FLAG IS NOT TO BE USED ON ANY S12X PAGING, BASED ON THE EXISTING GLOBAL ADDRESS MODE.**

### Cvdwarf Option usage (cont.)

Option	Description
<b>-so</b>	add stack offset. This option has to be selected when using debuggers using the SP value directly.  <b>THIS FLAG IS CURRENTLY ONLY MEANINGFULL FOR THE HC08/HCS08.</b>
<b>-v</b>	select verbose mode. <i>cvdwarf</i> will display information about its activity.

### Return Status

*cvdwarf* returns success if no problems are encountered. Otherwise it returns failure.

### Examples

Under MS/DOS, the command could be:

```
cvdwarfC:\test\acia.sm8
```

and will produce: C:\test\acia.elf

and the following command:

```
cvdwarf -o file C:\test\acia.sm8
```

will produce: file

Under UNIX, the command could be:

```
cvdwarf /test/acia.sm8
```

and will produce: test/acia.elf

# Compiler Error Messages

---

This appendix lists the error messages that the compiler may generate in response to errors in your program, or in response to problems in your host system environment, such as inadequate space for temporary intermediate files that the compiler creates.

The first pass of the compiler generally produces all user diagnostics. This pass deals with # control lines and lexical analysis, and then with everything else having to do with semantics. Only machine-dependent extensions are diagnosed in the code generator pass. If a pass produces diagnostics, later passes will not be run.

Any compiler message containing an exclamation mark **!** or the word '**PANIC**' indicates that the compiler has detected an inconsistent internal state. Such occurrences are uncommon and should be reported to the maintainers.

- [Parser \(cpstm8\) Error Messages](#)
- [Code Generator \(cgstm8\) Error Messages](#)
- [Assembler \(castm8\) Error Messages](#)
- [Linker \(clnk\) Error Messages](#)

## Parser (cpstm8) Error Messages

**<name> not a member** - field name not recognized for this struct/union

**<name> not an argument** - a declaration has been specified for an argument not specified as a function parameter

**<name> undefined** - a function or a variable is never defined

**FlexLM <message>**- an error is detected by the license manager

**\_asm string too long** - the string constant passed to *\_asm* is larger than 255 characters

**ambiguous space modifier** - a space modifier attempts to redefine an already specified modifier

**array size unknown** - the *sizeof* operator has been applied to an array of unknown size

**bad # argument in macro <name>** - the argument of a # operator in a *#define* macro is not a parameter

**bad # directive: <name>** - an unknown *#directive* has been specified

**bad # syntax** - # is not followed by an identifier

**bad ## argument in macro <name>** - an argument of a ## operator in a *#define* macro is missing

**bad #asm directive** - a *#asm* directive is not entered at a valid declaration or instruction boundary

**bad #define syntax** - a *#define* is not followed by an identifier

**bad #elif expression** - a *#elif* is not followed by a constant expression

**bad #else** - a *#else* occurs without a previous *#if*, *#ifdef*, *#ifndef* or *#elif*

**bad #endasm directive** - a *#endasm* directive is not closing a previous *#asm* directive

**bad #endif** - a *#endif* occurs without a previous *#if*, *#ifdef*, *#ifndef*, *#elif* or *#else*

**bad #if expression** - the expression part of a *#if* is not a constant expression

**bad #ifdef syntax** - extra characters are found after the symbol name

**bad #ifndef syntax** - extra characters are found after the symbol name

**bad #include syntax** - extra characters are found after the file name

**bad #pragma attribute directive** - syntax for the *#pragma attribute* directive is incorrect

**bad #pragma section directive** - syntax for the *#pragma section* directive is incorrect

**bad #pragma space directive** - syntax for the *#pragma space* directive is incorrect

**bad #pragma unroll directive** - syntax for the *#pragma unroll* directive is incorrect

**bad #undef syntax** - *#undef* is not followed by an identifier

**bad \_asm() argument type** - the first argument passed to *\_asm* is missing or is not a character string

**bad alias expression** - alias definition is not a valid expression

**bad alias value** - alias definition is not a constant expression

**bad bit number** - a bit number is not a constant between 0 and 7

**bad character <character>** - *<character>* is not part of a legal token

**bad defined syntax** - the *defined* operator must be followed by an identifier, or by an identifier enclosed in parenthesis

**bad function declaration** - function declaration has not been terminated by a right parenthesis

**bad integer constant** - an invalid integer constant has been specified

**bad invocation of macro <name>** - a *#define* macro defined without arguments has been invoked with arguments

**bad macro argument** - a parameter in a *#define* macro is not an identifier

**bad macro argument syntax** - parameters in a *#define* macro are not separated by commas

**bad proto argument type** - function prototype argument is declared without an explicit type

**bad real constant** - an invalid real constant has been specified

**bad space modifier** - a modifier beginning with a *@* character is not followed by an identifier

**bad structure for return** - the structure for return is not compatible with that of the function

**bad struct/union operand** - a structure or an union has been used as operand for an arithmetic operator

**bad symbol definition** - the syntax of a symbol defined by the **-d** option on the command line is not valid

**bad void argument** - the type *void* has not been used alone in a prototyped function declaration

**can't create <name>** - file *<name>* cannot be created for writing

**can't open <name>** - file *<name>* cannot be opened for reading

**can't redefine macro <name>** - macro *<name>* has been already defined

**can't undef macro <name>** - a *#undef* has been attempted on a predefined macro



**compare out of range** - a comparison is detected as being always true or always false (**+strict**)

**const assignment** - a *const* object is specified as left operand of an assignment operator

**constant assignment in a test** - an assignment operator has been used in the test expression of an *if*, *while*, *do*, *for* statements or a conditional expression (**+strict**)

**duplicate #pragma attribute name <name>** - two objects have been declared with the same <name> in *#pragma attribute* directives

**duplicate case** - two *case* labels have been defined with the same value in the same *switch* statement

**duplicate default** - a *default* label has been specified more than once in a *switch* statement

**embedded usage of tag name <name>** - a structure/union definition contains a reference to itself.

**enum size unknown** - the range of an enumeration is not available to choose the smallest integer type

**exponent overflow in real** - the exponent specified in a real constant is too large for the target encoding

**file too large for label information** - the source file is producing too many labels in the code and debug parts for the coding restrictions

**float value too large for integer cast** - a float constant is too large to be casted in an integer (**+strict**)

**hexadecimal constant too large** - an hexadecimal constant is too large to be represented on an integer

**illegal storage class** - storage class is not legal in this context

**illegal type specification** - type specification is not recognizable

**illegal void operation** - an object of type *void* is used as operand of an arithmetic operator

**illegal void usage** - an object of type *void* is used as operand of an assignment operator

**implicit int type in argument declaration** - an argument has been declared without any type (**+strict**)

**implicit int type in global declaration** - a global variable has been declared without any type (**+strict**)

**implicit int type in local declaration** - a local variable has been declared without any type (**+strict**)

**implicit int type in struct/union declaration** - a structure or union field has been declared without any type (**+strict**)

**incompatible argument type** - the actual argument type does not match the corresponding type in the prototype

**incompatible compare type** - operands of comparison operators must be of scalar type

**incompatible operand types** - the operands of an arithmetic operator are not compatible

**incompatible pointer assignment** - assigned pointers must have the same type, or one of them must be a pointer to *void*

**incompatible pointer operand** - a scalar type is expected when operators `+=` and `-=` are used on pointers

**incompatible pointer operation** - pointers are not allowed for that kind of operation

**incompatible pointer types** - the pointers of the assignment operator must be of equal or coercible type

**incompatible return type** - the return expression is not compatible with the declared function return type

**incompatible struct/union assignment** - a structure or an union has been used as operand for an assignement operator and the other operand is not a structure or an union

**incompatible struct/union operation** - a structure or an union has been used as operand of an arithmetic operator

**incompatible types in struct/union assignment** - structure or union types must be identical for assignment

**incomplete #elif expression** - a *#elif* is followed by an incomplete expression

**incomplete #if expression** - a *#if* is followed by an incomplete expression

**incomplete type** - structure type is not followed by a tag or definition

**incomplete type for debug information** - a structure or union is not completely defined in a file compiled with the debug option set

**integer constant too large** - a decimal constant is too large to be represented on an integer

**invalid #pragma attribute syntax** - a syntax error has been detected in a *#pragma attribute* directive

**invalid ? test expression** - the first expression of a ternary operator (*? :*) is not a testable expression

**invalid address expression** - the “address of” operator has been applied to a rvalue expression

**invalid address operand** - the “address of” operator has been applied to a *register* variable

**invalid address type** - the “address of” operator has been applied to a bitfield

**invalid alias** - an alias has been applied to an *extern* object

**invalid arithmetic operand** - the operands of an arithmetic operator are not of the same or coercible types

**invalid array dimension** - an array has been declared with a dimension which is not a constant expression

**invalid binary number** - the syntax for a binary constant is not valid

**invalid bit assignment** - the expression assigned to a bit variable must be scalar

**invalid bit initializer** - the expression initializing a bit variable must be scalar

**invalid bitfield size** - a bitfield has been declared with a size larger than its type size

**invalid bitfield type** - a type other than *int*, *unsigned int*, *char*, *unsigned char* has been used in a bitfield.

**invalid break** - a break may be used only in *while*, *for*, *do*, or *switch* statements

**invalid case** - a *case* label has been specified outside of a *switch* statement

**invalid case operand** - a case label has to be followed by a constant expression

**invalid cast operand** - the operand of a *cast* operator is not an expression

**invalid cast type** - a cast has been applied to an object that cannot be coerced to a specific type

**invalid conditional operand** - the operands of a conditional operator are not compatible

**invalid constant expression** - a constant expression is missing or is not reduced to a constant value

**invalid continue** - a continue statement may be used only in *while*, *for*, or *do* statements

**invalid default** - a *default* label has been specified outside of a *switch* statement

**invalid do test type** - the expression of a *do ... while()* instruction is not a testable expression

**invalid expression** - an incomplete or ill-formed expression has been detected

**invalid external initialization** - an external object has been initialized

**invalid floating point operation** - an invalid operator has been applied to floating point operands

**invalid for test type** - the second expression of a *for(;;)* instruction is not a testable expression

**invalid function member** - a function has been declared within a structure or an union

**invalid function type** - the function call operator *()* has been applied to an object which is not a function or a pointer to a function

**invalid if test type** - the expression of an *if ()* instruction is not a testable expression

**invalid indirection operand** - the operand of unary *\** is not a pointer

**invalid line number** - the first parameter of a *#line* directive is not an integer

**invalid local initialization** - the initialization of a local object is incomplete or ill-formed

**invalid lvalue** - the left operand of an assignment operator is not a variable or a pointer reference

**invalid narrow pointer cast** - a cast operator is attempting to reduce the size of a pointer

**invalid operand type** - the operand of a unary operator has an incompatible type

**invalid pointer cast operand** - a cast to a function pointer has been applied to a pointer that is not a function pointer

**invalid pointer initializer** - initializer must be a pointer expression or the constant expression 0

**invalid pointer operand** - an expression which is not of integer type has been added to a pointer

**invalid pointer operation** - an illegal operator has been applied to a pointer operand

**invalid pointer types** - two incompatible pointers have been subtracted

**invalid shift count type** - the right expression of a shift operator is not an integer

**invalid sizeof operand type** - the *sizeof* operator has been applied to a function

**invalid space for argument <name>** - an argument has been declared with a space modifier incompatible with the stack allocation

**invalid space for function** - a function has been declared with a space modifier incompatible with the function allocation

**invalid space for local <name>** - a local variable has been declared with a space modifier incompatible with the stack allocation

**invalid storage class** - storage class is not legal in this context

**invalid struct/union operation** - a structure or an union has been used as operand of an arithmetic operator

**invalid switch test type** - the expression of a *switch ()* instruction must be of integer type

**invalid typedef usage** - a typedef identifier is used in an expression

**invalid void pointer** - a *void* pointer has been used as operand of an addition or a subtraction

**invalid while test type** - the expression of a *while ()* instruction is not a testable expression

**misplaced #pragma section directive** - a *#pragma section* directive has been placed inside the body of a C function

**misplaced #pragma attribute name** - a *#pragma attribute* directive is not declaring any object

**missing ## argument in macro <name>** - an argument of a *##* operator in a *#define* macro is missing

**missing '>' in #include** - a file name of a *#include* directive begins with '<' and does not end with '>'

**missing ) in defined expansion** - a '(' does not have a balancing ')' in a *defined* operator

**missing ; in argument declaration** - the declaration of a function argument does not end with ';'.

**missing ; in local declaration** - the declaration of a local variable does not end with ';'.

**missing ; in member declaration** - the declaration of a structure or union member does not end with ';'.

**missing ? test expression** - the test expression is missing in a ternary operator (*? :*)

**missing \_asm() argument** - the *\_asm* function needs at least one argument

**missing argument** - the number of arguments in the actual function call is less than that of its prototype declaration

**missing argument for macro <name>** - a macro invocation has fewer arguments than its corresponding declaration

**missing argument name** - the name of an argument is missing in a prototyped function declaration

**missing array subscript** - an array element has been referenced with an empty subscript

**missing do test expression** - a *do ... while ()* instruction has been specified with an empty *while* expression

**missing enumeration member** - a member of an enumeration is not an identifier

**missing explicit return** - a return statement is not ending a non-void function (**+strict**)

**missing exponent in real** - a floating point constant has an empty exponent after the 'e' or 'E' character

**missing expression** - an expression is needed, but none is present

**missing file name in #include** - a *#include* directive is used, but no file name is present

**missing goto label** - an identifier is needed after a *goto* instruction

**missing if test expression** - an *if ()* instruction has been used with an empty test expression

**missing initialization expression** - a local variable has been declared with an ending '=' character not followed by an expression

**missing initializer** - a simple object has been declared with an ending '=' character not followed by an expression

**missing line number** - a line number is missing in a *#line* directive



**missing local name** - a local variable has been declared without a name

**missing member declaration** - a structure or union has been declared without any member

**missing member name** - a structure or union member has been declared without a name

**missing name in declaration** - a variable has been declared without a name

**missing prototype** - a function has been used without a fully prototyped declaration (**+strict**)

**missing prototype for inline function** - an inline function has been declared without a fully prototyped syntax

**missing return expression** - a simple return statement is used in a non-void function (**+strict**)

**missing switch test expression** - an expression in a *switch* instruction is needed, but is not present

**missing while** - a '*while*' is expected and not found

**missing while test expression** - an expression in a *while* instruction is needed, but none is present

**missing :** - a ':' is expected and not found

**missing ;** - a ';' is expected and not found. The parser reports such an error on the previous element as most of the time the ; is missing at the end of the declaration. When this error occurs on top of a file or just after a file include, the line number reported may not match the exact location where the problem is detected.

**missing (** - a '(' is expected and not found

**missing )** - a ')' is expected and not found

**missing ]** - a ']' is expected and not found

**missing {** - a '{' is expected and not found

**missing }** - a '}' is expected and not found

**missing } in enum definition** - an enumeration list does not end with a '}' character

**missing } in struct/union definition** - a structure or union member list does not end with a '}' character

**redeclared #pragma attribute name <name>** - a *#pragma attribute* object is already declared by another *#pragma attribute* directive

**redeclared argument <name>** - a function argument has conflicting declarations

**redeclared enum member <name>** - an *enum* element is already declared in the same scope

**redeclared external <name>** - an *external* object or function has conflicting declarations

**redeclared local <name>** - a *local* is already declared in the same scope

**redeclared proto argument <name>** - an identifier is used more than once in a prototype function declaration

**redeclared typedef <name>** - a *typedef* is already declared in the same scope

**redefined alias <name>** - an *alias* has been applied to an already declared object

**redefined label <name>** - a *label* is defined more than once in a function

**redefined member <name>** - an identifier is used more than once in structure member declaration

**redefined tag <name>** - a *tag* is specified more than once in a given scope

**repeated type specification** - the same type modifier occurs more than once in a type specification

**scalar type required** - type must be integer, floating, or pointer

**shift count out of range** - a constant shift count is larger than the shifted object size (**+strict**)

**size unknown** - an attempt to compute the size of an unknown object has occurred

**space attribute conflict** - a space modifier attempts to redefine an already specified modifier

**space conflict with #pragma attribute** - a space modifier declared with a *#pragma attribute* mismatches the space modifier specified in the object declaration

**stack attribute conflict on cast** - a cast is attempting to change the **@stack/@nostack** attribute of an object (**+strict**)

**string too long** - a string is used to initialize an array of characters shorter than the string length

**struct/union size unknown** - an attempt to compute a structure or union size has occurred on an undefined structure or union

**syntax error** - an unexpected identifier has been read

**token overflow** - an expression is too complex to be parsed

**too many argument** - the number of actual arguments in a function declaration does not match that of the previous prototype declaration

**too many arguments for macro <name>** - a macro invocation has more arguments than its corresponding macro declaration

**too many initializers** - initialization is completed for a given object before initializer list is exhausted

**too many spaces modifiers** - too many different names for '@' modifiers are used

**truncating assignment** - the right operand of an assignment is larger than the left operand (**+strict**)

**truncating constant cast** - a cast is attempting to narrow down the value of a constant (**+strict**)

**unbalanced ‘** - a character constant does not end with a single quote

**unbalanced “** - a string constant does not end with a double quote

**<name> undefined** - an undeclared identifier appears in an expression

**undefined label <name>** - a label is never defined

**undefined struct/union** - a structure or union is used and is never defined

**unexpected end of file** - last declaration is incomplete

**unexpected return expression** - a return with an expression has been used within a *void* function

**unknown enum definition** - an enumeration has been declared with no member

**unknown structure** - an attempt to initialize an undefined structure has been done

**unknown union** - an attempt to initialize an undefined union has been done

**unreachable code** - a code sequence cannot be accessed (**+strict**)

**value out of range** - a constant is assigned to a variable too small to represent its value (**+strict**)

**variable arguments in nostack mode** - a function has been declared with the ... syntax and the **@nostack** modifier (**+strict**)

**zero divide** - a divide by zero was detected

**zero modulus** - a modulus by zero was detected

## Code Generator (cgstm8) Error Messages

**bad builtin** - the *@builtin* type modifier can be used only on functions

**bad @interrupt usage** - the *@interrupt* type modifier can only be used on functions.

**invalid indirect call** - a function has been called through a pointer with more than one *char* or *int* argument, or is returning a structure.

**redefined space** - the version of *cpstm8* you used to compile your program is incompatible with *cgstm8*.

**unknown space** - you have specified an invalid space modifier *@xxx*

**unknown space modifier** - you have specified an invalid space modifier *@xxx*

**PANIC ! bad input file** - cannot read input file

**PANIC ! bad output file** - cannot create output file

**PANIC ! can't write** - cannot write output file

All other **PANIC !** messages should never happen. If you get such a message, please report it with the corresponding source program to COSMIC.

## Assembler (castm8) Error Messages

The following error messages may be generated by the assembler. Note that the assembler's input is machine-generated code from the compiler. Hence, it is usually impossible to fix things 'on the fly'. The problem must be corrected in the source, and the offending program(s) recompiled.

**bad .source directive** - a *.source* directive is not followed by a string giving a file name and line numbers

**bad addressing mode** - an invalid addressing mode have been constructed

**bad argument number**- a parameter sequence *n* uses a value negative or greater than 9

**bad character constant** - a character constant is too long for an expression

**bad comment delimiter**- an unexpected field is not a comment

**bad constant** - a constant uses illegal characters

**bad else** - an *else* directive has been found without a previous *if* directive

**bad endif** - an *endif* directive has been found without a previous *if* or *else* directive

**bad file name** - the *include* directive operand is not a character string

**bad index register** - an invalid register has been used in an indexed addressing mode

**bad register** - an invalid register has been specified as operand of an instruction

**bad relocatable expression** - an external label has been used in either a constant expression, or with illegal operators

**bad string constant** - a character constant does not end with a single or double quote

**bad symbol name: <name>** - an expected symbol is not an identifier

**can't create <name>** - the file <name> cannot be opened for writing

**can't open <name>** - the file <name> cannot be opened for reading

**can't open source <name>** - the file <name> cannot be included

**cannot include from a macro** - the directive *include* cannot be specified within a macro definition

**cannot move back current pc** - an *org* directive has a negative offset

**illegal size** - the size of a *ds* directive is negative or zero

**missing label** - a label must be specified for this directive

**missing operand** - operand is expected for this instruction

**missing register** - a register is expected for this instruction

**missing string** - a character string is expected for this directive

**relocatable expression not allowed** - a constant is needed

**section name <name> too long** - a section name has more than 15 characters

**string constant too long** - a string constant is longer than 255 characters

**symbol <name> already defined** - attempt to redefine an existing symbol

**symbol <name> not defined** - a symbol has been used but not declared

**syntax error** - an unexpected identifier or operator has been found

**too many arguments** - a macro has been invoked with more than 9 arguments

**too many back tokens** - an expression is too complex to be evaluated

**unclosed if** - an *if* directive is not ended by an *else* or *endif* directive

**unknown instruction** <name> - an instruction not recognized by the processor has been specified

**value too large** - an operand is too large for the instruction type

**zero divide** - a divide by zero has been detected



## Linker (clnk) Error Messages

**-a not allowed with -b or -o** - the *after* option cannot be specified if any start address is specified.

**+def symbol <symbol> multiply defined** - the symbol defined by a *+def* directive is already defined.

**bad address (<value>) for zero page symbol <name>** - a symbol declared in the zero page is allocated to an address larger than 8 bits.

**bad file format** - an input file has not an object file format.

**bad number in +def** - the number provided in a *+def* directive does not follow the standard C syntax.

**bad number in +spc <segment>** - the number provided in a *+spc* directive does not follow the standard C syntax.

**bad processor type** - an object file has not the same configuration information than the others.

**bad reloc code** - an object file contains unexpected relocation information.

**bad section name in +def** - the name specified after the '@' in a *+def* directive is not the name of a segment.

**can't create map file <file>** - map file cannot be created.

**can't create <file>** - output file cannot be created.

**can't locate .text segment for initialization** - initialized data segments have been found but no host segment has been specified.

**can't locate shared segment** - shared datas have been found but no host segment has been specified.

**can't open file <file>** - input file cannot be found.

**file already linked** - an input file has already been processed by the linker.

**function <function> is recursive** - a *nostack* function has been detected as recursive and cannot be allocated.

**function <function> is reentrant** - a function has been detected as reentrant. The function is both called in an interrupt function and in the main code.

**incomplete +def directive** - the **+def** directive syntax is not correct.

**incomplete +seg directive** - the **+seg** directive syntax is not correct.

**incomplete +spc directive** - the **+spc** directive syntax is not correct.

**init segment cannot be initialized** - the host segment for initialization cannot be itself initialized.

**invalid @ argument** - the syntax of an optional input file is not correct.

**invalid -i option** - the **-i** directive is followed by an unexpected character.

**missing command file** - a link command file must be specified on the command line.

**missing output file** - the **-o** option must be specified.

**missing '=' in +def** - the **+def** directive syntax is not correct.

**missing '=' in +spc <segment>** - the **+spc** directive syntax is not correct.

**named segment <segment> not defined** - a segment name does not match already existing segments.

**no default placement for segment <segment>** - a segment is missing **-a** or **-b** option.

**prefixed symbol <name> in conflict** - a symbol beginning by 'f\_' (for a banked function) also exists without the 'f' prefix.

**read error** - an input object file is corrupted

**segment <segment> and <segment> overlap** - a segment is overlapping an other segment.

**segment <segment> size overflow** - the size of a segment is larger than the maximum value allowed by the **-m** option.

**shared segment not empty** - the host segment for shared data is not empty and cannot be used for allocation.

**symbol <symbol> multiply defined** - an object file attempts to redefine a symbol.

**symbol <symbol> not defined** - a symbol has been referenced but never defined.

**unknown directive** - a directive name has not been recognized as a linker directive.



# Modifying Compiler Operation

---

This chapter tells you how to modify compiler operation by making changes to the standard configuration file. It also explains how to create your own programmable options” which you can use to modify compiler operation from the [cxstm8.cxf](#).

## The Configuration File

The configuration file is designed to define the default options and behaviour of the compiler passes. It will also allow the definition of programmable options thus simplifying the compiler configuration. A configuration file contains a list of options similar to the ones accepted for the compiler driver utility **cxstm8**.

These options are described in **Chapter 4**, “[Using The Compiler](#)”. There are two differences: the option **-f** cannot be specified in a configuration file, and the extra **-m** option has been added to allow the definition of a programmable compiler option, as described in the next paragraph.

The contents of the configuration file **cxstm8.cxf** as provided by the default installation appears below:

```
# CONFIGURATION FILE FOR STM8 COMPILER
# Copyright (c) 2008 by COSMIC Software
#
-pu                # unsigned char
-ppb               # pack local bit variables
-i c:\cx32\hstm8  # include path
#
-m debug:x         # debug: produce debug info
-m fast:i,,,compact # fast: inline long transferts
-m compact:,,f7    # compact: do not factorize code
-m nobss:bss       # nobss: do not use bss
-m nocst:ct        # nocst: constant in text section
-m nocross:nc      # functions do not cross boundaries
-m proto:p         # proto: enable prototype checking
-m rev:rb          # rev: reverse bit field order
-m strict:ck       # strict: enforce type checking
-m split:sf        # functions in different sections
-m mods:hmods.h,fl # stack model
-m modsl:hmodsl.h,fl # stack long model
-m mods0:hmods0.h  # stack model 64K
-m modsl0:hmodsl0.h # stack long model 64K
-m warn:w1         # warn: enable warnings
```

The following command line:

```
cxstm8 hello.c
```

in combination with the above configuration file directs the **cxstm8** compiler to execute the following commands:

```
cpstm8 -o \2.cx1 -u -i\cosmic\hstm8 hello.c
cgstm8 -o \2.cx2 \2.cx1
costm8 -o \2.cx1 \2.cx2
castm8 -o hello.o -i\cosmic\hstm8 \2.cx1
```

## Changing the Default Options

To change the combination of options that the compiler will use, edit the configuration file and add your specific options using the **-p** (for the **p**arser), **-g** (for the code **g**enerator), **-o** (for the **o**ptimizer) and **-a** (for the **a**sssembler) options. If you specify an invalid option or combination of options, compilation will not proceed beyond the step where the error occurred. You may define up to 128 such options.

### Creating Your Own Options

To create a programmable option, edit the configuration file and define the parametrable option with the **-m\*** option. The string *\** has the following format:

```
name:popt,gopt,oopt,aopt,exclude...
```

The first field defines the option *name* and must be ended by a colon character ':'. The four next fields describe the effect of this option on the four passes of the compiler, respectively the *parser*, the *generator*, the *optimizer* and the *assembler*. These fields are separated by a comma character ','. If no specific option is needed on a pass, the field has to be specified empty. The remaining fields, if specified, describe a exclusive relationship with other defined options. If two *exclusive* options are specified on the command line, the compiler will stop with an error message. You may define up to 128 programmable options. At least one field has to be specified. Empty fields need to be specified only if a useful field has to be entered after.

In the following example:

```
-m dl1:1,dl1,,,dl2# dl1: line option 1  
-m dl2:1,dl2,,,dl1# dl1: line option 2
```

the two options *dl1* and *dl2* are defined. If the option **+dl1** is specified on the compiler command line, the specific option **-l** will be used for the *parser* and the specific option **-dl1** will be used for the code *generator*. No specific option will be used for the *optimizer* and for the *assembler*. The option *dl1* is also declared to be exclusive with the option *dl2*, meaning that *dl1* and *dl2* will not be allowed together on the compiler command line. The option *dl2* is defined in the same way.



## Example

The following command line

```
cxstm8 +nobss hello.c
```

in combination with the previous configuration file directs the **cxstm8** compiler to execute the following commands:

```
cpstm8 -o \2.cx1 -u -i\cosmic\hstm8 hello.c
cgstm8 -o \2.cx2 -bss \2.cx1
costm8 -o \2.cx1 \2.cx2
castm8 -o hello.o -i\cosmic\hstm8 \2.cx1
```



# STM8 Machine Library

---

This appendix describes each of the functions in the Machine Library (**libm**, using **d\_** prefix and **libm0** for application smaller than 64K, using **c\_** prefix). These functions provide the interface between the **STM8** microcontroller hardware and the functions required by the code generator. They are described in reference form, and listed alphabetically.

Note that machine library functions handle values as follows:

- **integer** in the registers **x** or **y**.
- **longs** and **floats** in the four byte memory location **c\_lreg**, (“float register” or “long register” depending on context).
- **pointer to long** or **float** in registers **x** or **y** for **@near** pointers and in **x** and the memory location **c\_x** or **y** and the memory location **c\_y** for **@far** pointers.

The library functions using a pointer to far memory have a name beginning with the ‘**f**’ letter, and the pointer is located in the pair composed by the **x** register for the lower word, and the memory location **c\_x** for the upper byte. The following describes only the function handling data in near memory. Their equivalent functions have the same description except for the pointer location and size.

# c\_bitfw

## Description

Update an int bitfield in near memory

## Syntax

```
; bitfield address in y  
; mask in c_x and c_x+1  
; value in a and xl  
call c_bitfw
```

## Function

*c\_bitfw* is used to update a 16 bits bitfield located in extended memory by a new value located in the **a** and **xl** register pair. The value loaded from extended memory is first and'ed with the mask located in the **c\_x** memory location. It is then or'ed with the value in the **a** and **xl** register pair and stored back in memory.

## Return Value

None.

# c\_ewbfb

## Description

Eeprom char bit field update

## Syntax

```
; value in x  
; address in c_x and extension  
; mask in a  
    call c_ewbfb
```

## Function

*c\_ewbfb* updates a char bit field (8 bits sized) located in *eeprom* with a new value. The new value is in register **x** and is right justified. The byte address in *eeprom* is in **c\_x** and **c\_x+1**, and the mask, giving the bit field size and location, is in register **a**. The function waits for the time necessary to program the new value.

## See Also

*c\_ewstr*

# c\_ewrc

## Description

Write a char in *eprom*

## Syntax

```
; value in a  
; address in x  
call c_ewrc
```

## Function

*c\_ewrc* writes a byte in *eprom*. The new byte value is in the **a** register and its address in *eprom* is in **x**. The function tests if the erasure is necessary, and do it only in that case. Then if the new value is different from one in *eprom*, the new byte is programmed. The function waits for the time necessary to program correctly the byte. The function does not test if the byte address is in the address range corresponding to the existing *eprom*.

## See Also

*c\_ewrl*, *c\_ewrw*

# `c_ewrl`

## Description

Write a long int in *eeeprom*

## Syntax

```
; value in c_lreg  
; address in x  
    call c_ewrl
```

## Function

`c_ewrl` writes a long int in *eeeprom*. The new value is in the long register, and its address in *eeeprom* is in **x**. Each byte is programmed independently by the `c_ewrc` function.

## See Also

`c_ewrc`, `c_ewrw`

## c\_eevrw

### Description

Write a short int in *eeprw*

### Syntax

```
; value on the stack  
; address in x  
call c_eevrw
```

### Function

*c\_eevrw* writes a short int in *eeprw*. The new value is on the stack, and its address in *eeprw* is in **x**. Each byte is programmed independently by the *c\_eevrc* function.

### See Also

*c\_eevrc*, *c\_eevrl*



# c\_ewstr

## Description

Move a structure in *eeeprom*

## Syntax

```
; source address in y  
; destination address in x  
; size in a  
call c_ewstr
```

## Function

*c\_ewstr* moves a structure into an *eeeprom* memory location. Pointer to source is in **y**, and pointer to destination is in **x**. The structure size is in register **a**. Each byte is programmed independently by the *c\_ewrc* function.

## See Also

*c\_ewbfb*, *c\_ewrc*

# c\_fadd

## Description

Add float to float

## Syntax

```
; left in float register  
; pointer to right in x register  
    call c_fadd  
; result in float register
```

## Function

*c\_fadd* adds the float in *float register* to the float indicated by the **x** register. No check is made for overflow.

## Return Value

The resulting value is in *float register*. Flags have no meaningful value upon return.

## See Also

*c\_fsub*

# c\_fcmp

## Description

Compare floats

## Syntax

```
; left in float register  
; pointer to right in x register  
    call c_fcmp  
; result in flags
```

## Function

*c\_fcmp* compares the float in *float register* with the float pointered at by the **x** register.

## Return Value

The **N** and **Z** flags are set to reflect the value (left-right).

# c\_fdiv

## Description

Divide float by float

## Syntax

```
; left in float register  
; pointer to right in x register  
    call c_fdiv  
; result in float register
```

## Function

*c\_fdiv* divides the float in *float register* by the float pointered at by the **x** register.

## Return Value

The resulting value is in *float register*. Flags have no meaningful value upon return.

# c\_fgadd

## Description

Add float to float in memory

## Syntax

```
; pointer to left in x register  
; right in float register  
    call c_fgadd  
; result in memory
```

## Function

*c\_fgadd* adds the float in the float pointered at by the **x** register to the float register.

## Return Value

The resulting value is stored at the location pointered at by the **x** register, meaning that the left operand is updated. Flags have no meaningful value upon return.

## See Also

*c\_fgsub*

# c\_fgmul

## Description

Multiply float by float in memory

## Syntax

```
; pointer to left in x register  
; right in float register  
    call c_fgmul  
; result in memory
```

## Function

*c\_fgmul* multiplies the float in *float register* by the float pointered at by the **x** register.

## Return Value

The resulting value is stored at the location pointered at by **x**. Flags have no meaningful value upon return.

# c\_fgsub

## Description

Subtract float from float in memory

## Syntax

```
; pointer to left in x register  
; right in float register  
    call c_fgsub  
; result in memory
```

## Function

*c\_fgsub* subtracts the float pointed at by the **x** register from the float in *float register*. No check is made for overflow.

## Return Value

The resulting value is stored at the location pointed at by **x**. Flags have no meaningful value upon return.

## See Also

*c\_fgadd*

# c\_fmul

## Description

Multiply float by float

## Syntax

```
; left in float register  
; pointer to right in x register  
    call c_fmul  
; result in float register
```

## Function

*c\_fmul* multiplies the float in *float register* by the float pointered at by the **x** register.

## Return Value

The resulting value is in *float register*. Flags have no meaningful value upon return



# c\_fneg

## Description

Negate a float

## Syntax

```
; operand in float register
    call c_fneg
; result in operand
```

## Function

*c\_fneg* negates the float pointered at by the *float register*.

## Return Value

The resulting value is in *float register*. Flags have no meaningful value upon return.

# c\_fsub

## Description

Subtract float from float

## Syntax

```
; left in float register  
; pointer to right in x register  
    call c_fsub  
; result in float register
```

## Function

*c\_fsub* subtracts the float pointed to by the **x** register from the float in *float register*. No check is made for overflow.

## Return Value

The resulting value is in *float register*. Flags have no meaningful value upon return.

## See Also

*c\_fadd*

# c\_ftoi

## Description

Convert float to integer

## Syntax

```
; float in float register  
    call c_ftoi  
; result in register x
```

## Function

*c\_ftoi* converts the float in *float register* to a two byte integer in the **x** register. No check is made for overflow.

## Return Value

The resulting value is in **x**. Flags have no meaningful value upon return.

## See Also

*c\_ftol*, *c\_itof*, *c\_itol*, *c\_ltof*

# c\_ftol

## Description

Convert float into long integer

## Syntax

```
; float in float register  
    call c_ftol  
;result in long register
```

## Function

*c\_ftol* converts the float in *float register* to a four byte integer in *long register*. No check is made for overflow.

## Return Value

The resulting value is in *long register*. Flags have no meaningful value upon return.

## See Also

*c\_ftoi, c\_itof, c\_itol, c\_ltof*

# c\_fzmp

## Description

Compare a float in memory to zero

## Syntax

```
; pointer to operand in x register  
    call c_fzmp  
; result in flags
```

## Function

*c\_fzmp* compares the float pointered by the **x** register against zero.

## Return Value

The **N** and **Z** flags are set to reflect the operand value.

# c\_getlx

## Description

Get a long word from memory

## Syntax

```
; long address in x  
    call c_getlx  
; result in long register
```

## Function

*c\_getlx* gets a long integer from memory using a pointer loaded in **x**.  
The result is left in the *long register*.

## Return Value

The byte is loaded in the *long register*. Flags have no meaningful value upon return.

## See Also

*c\_getly*, *c\_getwfx*, *c\_getwfy*

# c\_getly

## Description

Get a long word from memory

## Syntax

```
; long address in y  
    call c_getly  
; result in long register
```

## Function

*c\_getly* gets a long integer from memory using a pointer loaded in **y**.  
The result is left in the *long register*.

## Return Value

The byte is loaded in the *long register*. Flags have no meaningful value upon return.

## See Also

*c\_getlx*, *c\_getwfx*, *c\_getwfy*

# c\_getwfx

## Description

Get a word from far memory

## Syntax

```
; word address in x and c_x  
    call c_getwfx  
; result in a and xl
```

## Function

*c\_getwfx* gets a word from far memory using a pointer loaded in **x** and **c\_x**. The result is left in the **a** and **xl** registers.

## Return Value

The word is loaded in the **a** and **xl** registers. Flags have no meaningful value upon return.

## See Also

*c\_getlx*, *c\_getly*, *c\_getwfy*



# c\_getwfy

## Description

Get a word from far memory

## Syntax

```
; word address in y and c_y  
    call c_getwfy  
; result in a and x1
```

## Function

*c\_getwfy* gets a word from far memory using a pointer loaded in **y** and **c\_y**. The result is left in the **a** and **x1** registers.

## Return Value

The word is loaded in the **a** and **x1** registers. Flags have no meaningful value upon return.

## See Also

*c\_getlx*, *c\_getly*, *c\_getwfx*

# c\_getxfx

## Description

Get a word from far memory

## Syntax

```
; word address in x and c_x  
    call c_getxfx  
; result in x
```

## Function

*c\_getxfx* gets a word from far memory using a pointer loaded in **x** and **c\_x**. The result is left in the **x** register.

## Return Value

The word is loaded in the **x** register. Flags have no meaningful value upon return.

## See Also

*c\_getlx*, *c\_getly*, *c\_getwfy*

# c\_getxfy

## Description

Get a word from far memory

## Syntax

```
; word address in y and c_y  
    call c_getxfy  
; result in x
```

## Function

*c\_getxfy* gets a word from far memory using a pointer loaded in **y** and **c\_y**. The result is left in the **x** register.

## Return Value

The word is loaded in the **x** register. Flags have no meaningful value upon return.

## See Also

*c\_getlx*, *c\_getly*, *c\_getwfy*

# c\_getyfx

## Description

Get a word from far memory

## Syntax

```
; word address in x and c_x  
    call c_getyfx  
; result in y
```

## Function

*c\_getyfx* gets a word from far memory using a pointer loaded in **x** and **c\_x**. The result is left in the **y** register.

## Return Value

The word is loaded in the **y** register. Flags have no meaningful value upon return.

## See Also

*c\_getlx*, *c\_getly*, *c\_getwfy*

# c\_getyfy

## Description

Get a word from far memory

## Syntax

```
; word address in y and c_y  
    call c_getyfy  
; result in y
```

## Function

*c\_getyfy* gets a word from far memory using a pointer loaded in **y** and **c\_y**. The result is left in the **y** register.

## Return Value

The word is loaded in the **y** register. Flags have no meaningful value upon return.

## See Also

*c\_getlx*, *c\_getly*, *c\_getwfy*

# c\_idiv

## Description

Quotient of integer division

## Syntax

```
; dividend in x  
; divisor in y  
    call c_idiv  
; quotient in x
```

## Function

*c\_idiv* divides the two byte integer in the **x** register, by the two byte integer in the **y** register. Values are assumed to be signed. If division by zero is attempted, the result is the unchanged dividend.

## Return Value

The quotient is placed in **x**. Flags have no meaningful value upon return.

## See Also

*c\_udiv*

# c\_imul

## Description

Integer multiplication

## Syntax

```
; left in x
; right in y
    call c_imul
; result in x
```

## Function

*c\_imul* multiplies the two byte integer in the **x** register, by the two byte integer in the **y** register. No check is made for overflow.

## Return Value

The resulting value is in **x**. Flags have no meaningful value upon return

# c\_itof

## Description

Convert integer into float

## Syntax

```
; operand in x
    call c_itof
; result in float register
```

## Function

*c\_itof* converts the two byte integer in the **x** register, to a float stored in *float register*.

## Return Value

The resulting value is in *float register*. Flags have no meaningful value upon return.

## See Also

*c\_ltof*, *c\_ultof*, *c\_xtof*, *c\_uitof*, *c\_uxtof*



# c\_itol

## Description

Convert integer into long

## Syntax

```
; operand in a and x1  
    call c_itol  
; result in long register
```

## Function

*c\_itol* converts the two byte integer in the **a** and **x1** register pair, to a long integer stored in *long register*.

## Return Value

The resulting value is in *long register*. Flags have no meaningful value upon return.

## See Also

*c\_xtol*, *c\_uitol*, *c\_uxtol*

# c\_itolx

## Description

Convert integer into long

## Syntax

```
; operand in x  
    call c_itolx  
; result in long register
```

## Function

*c\_itolx* converts the two byte integer in the **x** register, to a long integer stored in *long register*.

## Return Value

The resulting value is in *long register*. Flags have no meaningful value upon return.

## See Also

*c\_xtol*, *c\_uitol*, *c\_uxtol*

# c\_itoly

## Description

Convert integer into long

## Syntax

```
; operand in y  
    call c_itoly  
; result in long register
```

## Function

*c\_itoly* converts the two byte integer in the **y** register, to a long integer stored in *long register*.

## Return Value

The resulting value is in *long register*. Flags have no meaningful value upon return.

## See Also

*c\_xtol*, *c\_uitol*, *c\_uxtol*

# c\_jctab

## Description

Perform C switch statement on char

## Syntax

```
; value in a  
; table address after the call  
call c_jctab
```

## Function

*c\_jctab* is called to switch to the proper code segment, depending on a value in the **a** register and an address table found just after the call instruction, and consisting in a list of two bytes signed offsets.

## Return Value

*c\_jctab* jumps to the proper code. It never returns.

## See Also

*c\_jstab*, *c\_jltab*

# c\_jltab

## Description

Perform C switch statement on long

## Syntax

```
; value in long register  
; table address in x  
jp c_jltab
```

## Function

*c\_jltab* is called to switch to the proper code segment, depending on a value and an address table. The top of the table is found in the **x** register, and consists of a count followed by a list of pairs. A pair consists of a value followed by an address. The pair list is ended by the default address. All values are four byte integers. All addresses and the count are two byte integers.

## Return Value

*c\_jltab* jumps to the proper code. It never returns.

## See Also

*c\_jctab*, *c\_jstab*

# c\_jstab

## Description

Perform C switch statement on integer

## Syntax

```
; value in x  
; table address after the call  
call c_jstab
```

## Function

*c\_jstab* is called to switch to the proper code segment, depending on a value in the **x** register and an address table found just after the call instruction, and consisting in a list of two bytes signed offsets.

## Return Value

*c\_jstab* jumps to the proper code. It never returns.

## See Also

*c\_jctab*, *c\_jltab*

# c\_ladc

## Description

Long integer addition

## Syntax

```
; left in long register  
; right in a register  
    call c_ladc  
; result in long register
```

## Function

*c\_ladc* adds the four byte integer in *long register* and the unsigned char in the **a** register.

## Return Value

The result is in *long register*. Flags are not meaningful upon return.

## See Also

*c\_ladd*

# c\_ladd

## Description

Long integer addition

## Syntax

```
; left in long register  
; pointer to right in x register  
    call c_ladd  
; result in long register
```

## Function

*c\_ladd* adds the four byte integer in *long register* and the four byte integer pointed at by the **x** register.

## Return Value

The result is in *long register*. Flags are not meaningful upon return.

## See Also

*c\_lcmp*, *c\_lsub*



# c\_land

## Description

Bitwise AND for long integers

## Syntax

```
; left in long register  
; pointer to right in x register  
    call c_land  
; result in long register
```

## Function

*c\_land* operates a bitwise AND between the operands. Each operand is taken to be a four byte integer.

## Return Value

The result is in *long register*. Flags are not meaningful upon return.

## See Also

*c\_lor*, *c\_lxor*

# c\_lcmp

## Description

Long integer compare

## Syntax

```
; left in long register  
; pointer to right in x register  
    call c_lcmp  
; result in long flags
```

## Function

*c\_lcmp* compares the four byte integer in *long register* to the four byte integer pointered by the **x** register.

## Return Value

Flags are set accordingly.

## See Also

*c\_ladd*, *c\_lsub*, *c\_lsmpl*

# c\_ldiv

## Description

Quotient of long integer division

## Syntax

```
; dividend in long register  
; pointer to divisor in x register  
    call c_ldiv  
; quotient in long register
```

## Function

*c\_ldiv* divides the four byte integer in *long register* by the four byte integer pointered by the **x** register. Values are assumed to be signed. If division by zero is attempted, the result is the unchanged dividend.

## Return Value

The quotient is in *long register* and the flags are not meaningful upon return.

## See Also

*c\_ludv*, *c\_lmod*, *c\_lumd*

# c\_lgadc

## Description

Long addition

## Syntax

```
; pointer to left in x register  
; right in a register  
    call c_lgadc  
; result in left
```

## Function

*c\_lgadc* performs the long addition of the unsigned char value in the **a** register and the long value pointed at by the **x** register.

## Return Value

The result is stored at the location in the **x** register. Flags are not meaningful upon return.

## See Also

*c\_lgadd*, *c\_pgadc*

# c\_lgadd

## Description

Long addition

## Syntax

```
; pointer to left in x register  
; right in long register  
    call c_lgadd  
; result in left
```

## Function

*c\_lgadd* performs the long addition of the value pointered by the **x** register and the value in *long register*.

## Return Value

The result is stored at the location pointered by the **x** register. Flags are not meaningful upon return.

## See Also

*c\_lgadc*

# c\_lgand

## Description

Long bitwise AND

## Syntax

```
; left in long register  
; pointer to right in x register  
    call c_lgand  
; result in left
```

## Function

*c\_lgand* performs the long bitwise AND of the value in *long register* and the value pointered by the **x** register.

## Return Value

The result is stored in *long register*. Flags are not meaningful upon return.

# c\_lglsh

## Description

Long shift left

## Syntax

```
; pointer to long in x register  
; shift count in a register  
    call c_lglsh  
; result in memory
```

## Function

*c\_lglsh* performs the long left shift of the value pointed by the **x** register by the bit count in the **a** register. No check is done against silly counts.

## Return Value

The result is stored in the location pointed by **x**. Flags are not meaningful upon return.

# c\_lgmul

## Description

Long multiplication in memory

## Syntax

```
; pointer to left in x register  
; right in long register  
    call c_lgmul  
; result in left
```

## Function

*c\_lgmul* performs the long multiplication of the value pointered by the **x** register, by the value in *long register*.

## Return Value

The result is stored in the location pointered by **x**. Flags are not meaningful upon return.

## See Also

*c\_lmul*



# c\_lneg

## Description

Negate a long integer in memory

## Syntax

```
; pointer to operand in x register
    call c_lneg
; result in memory
```

## Function

*c\_lneg* negates the four byte integer pointered by the **x** register.

## Return Value

The result is in the location pointered by **x**. The flags are not meaningful upon return.

## See Also

*c\_lneg*

# c\_lgor

## Description

Long bitwise OR

## Syntax

```
; pointer to left in x register  
; right in long register  
    call c_lgor  
; result in left
```

## Function

*c\_lgor* performs the long bitwise OR of the value in *long register* and the value pointered by the **x** register.

## Return Value

The result is stored in *long register*. Flags are not meaningful upon return.

# c\_lgrsh

## Description

Signed long shift right

## Syntax

```
; pointer to long in x register  
; shift count in a register  
    call c_lgrsh  
; result in memory
```

## Function

*c\_lgrsh* performs the signed long right shift of the value pointed to by the **x** register and the value in *long register*. No check is done against silly counts. Because the value is signed, arithmetic shift instructions are used.

## Return Value

The result is stored in the location pointed to by **x**. Flags are not meaningful upon return.

# c\_lgsbc

## Description

Long subtraction

## Syntax

```
; pointer to left in x register  
; right in a register  
    call c_lgsbc  
; result in left
```

## Function

*c\_lgsbc* evaluates the (long) difference between the value pointed by the **x** register and the unsigned char value in the **a** register.

## Return Value

The result is stored in the location pointed by **x**. Flags are not meaningful upon return.

## See Also

*c\_lgsub*

# c\_lgsub

## Description

Long subtraction

## Syntax

```
; pointer to left in x register  
; right in long register  
    call c_lgsub  
; result in left
```

## Function

*c\_lgsub* evaluates the (long) difference between the value pointed by the **x** register and the value in *long register*.

## Return Value

The result is stored in the location pointed by **x**. Flags are not meaningful upon return.

## See Also

*c\_lgsbc*

# c\_lgursh

## Description

Unsigned long shift right

## Syntax

```
; pointer to long in x register  
; shift count in a register  
    call c_lgursh  
; result in memory
```

## Function

*c\_lgursh* performs the unsigned long right shift of the value pointered by the **x** register and the value in *long register*. No check is done against silly counts. Because the value is unsigned, logical shift instructions are used.

## Return Value

The result is stored in the location pointered by **x**. Flags are not meaningful upon return.

# c\_lgxor

## Description

Long bitwise exclusive OR

## Syntax

```
; pointer to right in x register  
; left in long register  
    call c_lgxor  
; result in left
```

## Function

*c\_lgxor* performs the long bitwise Exclusive OR of the value in *long register* and the value pointered by the **x** register.

## Return Value

The result is stored in *long register*. Flags are not meaningful upon return.

# c\_llsh

## Description

Long integer shift left

## Syntax

```
; operand in long register  
; shift count in a register  
    call c_llsh  
; result in long register
```

## Function

*c\_llsh* shifts left four byte integer in *long register* by the number of places specified by the **r** register. A zero count leaves the *long register* unchanged. No check is made for invalid counts.

## Return Value

The resulting value is in *long register*. Flags are not meaningful upon return.

## See Also

*c\_lrsh*, *c\_lursh*



# c\_lmod

## Description

Remainder of long integer division

## Syntax

```
; left in long register  
; pointer to right in x register  
    call c_lmod  
; remainder in long register
```

## Function

*c\_lmod* divides the four byte integer in *long register* by the four byte integer pointed to by the **x** register. Values are assumed to be signed. The dividend is returned if a division by zero is attempted.

## Return Value

The remainder is stored in *long register*. Flags are not meaningful upon return.

## See Also

*c\_lumd*, *c\_ldiv*, *c\_udiv*

# c\_lmul

## Description

Multiply long integer by long integer

## Syntax

```
; left in long register  
; pointer to right in x register  
    call c_lmul  
; result in long register
```

## Function

*c\_lmul* multiplies the four byte integer in *long register* by the four byte integer pointered by the **x** register. No check is made for overflow.

## Return Value

The resulting value is in *long register*. Flags are not meaningful upon return.

## See Also

*c\_lgmul*

# c\_lneg

## Description

Negate a long integer

## Syntax

```
; operand in long register
    call c_lneg
; result in long register
```

## Function

*c\_lneg* negates the four byte integer in *long register*.

## Return Value

The result is in *long register*. The flags are not meaningful upon return.

## See Also

*c\_lneg*

# c\_lor

## Description

Bitwise OR with long integers

## Syntax

```
; left in long register  
; pointer to right in x register  
  call c_lor  
; result in long register
```

## Function

*c\_lor* operates a bitwise OR between the contents of *long register* and the long pointed by the **x** register. Each operand is taken to be a four byte integer.

## Return Value

The result is in *long register*. The flags are not meaningful upon return.

## See Also

*c\_land*, *c\_lxor*

# `c_lrsh`

## Description

Long integer right shift

## Syntax

```
; operand in long register  
; shift count in a register  
    call c_lrsh  
; result in long register
```

## Function

`c_lrsh` right shifts the four byte integer in *long register* by the number of bits specified by the `a` register. A zero count leaves the *long register* unchanged. No check is made for invalid counts. The value is assumed to be signed, so a negative value will stay negative as by an arithmetic shift.

## Return Value

The resulting value stays in *long register*. Flags are not meaningful upon return.

## See Also

`c_llsh`, `c_lursh`

# c\_lrzmp

## Description

Long test against zero

## Syntax

```
; operand in long register  
    call c_lrzmp  
; result in the flags
```

## Function

*c\_lrzmp* tests the value in the *long register* and updates the sign and zero flags.

## Return Value

Nothing, but the flags.

## See Also

*c\_lzmp*

# c\_lsb

## Description

Long integer subtraction

## Syntax

```
; left in long register  
; right in a register  
    call c_lsb  
; result in long register
```

## Function

*c\_lsb* subtracts the unsigned char in the **a** register from the four byte integer in *long register*.

## Return Value

The result is in *long register*. Flags are not meaningful upon return.

## See Also

*c\_lsub*

# c\_lsub

## Description

Long integer subtraction

## Syntax

```
; left in long register  
; pointer to right in x register  
  call c_lsub  
; result in long register
```

## Function

*c\_lsub* subtracts the four byte integer pointed to by the **x** register from the four byte integer in *long register*.

## Return Value

The result is in *long register*. Flags are not meaningful upon return.

## See Also

*c\_ladd*, *c\_lcmp*



# c\_lsm

## Description

Long integer compare with overflow

## Syntax

```
; left in long register  
; pointer to right in x register  
    call c_lsm  
; result in long flags
```

## Function

*c\_lsm* compares the four byte integer in *long register* to the four byte integer pointered by the **x** register and updates the resulting flags in case of overflow.

## Return Value

Flags are set accordingly.

## See Also

*c\_ladd*, *c\_lsub*, *c\_lcmp*

# c\_ltof

## Description

Convert long integer into float

## Syntax

```
; operand in float integer  
    call c_ltof  
; result in float register
```

## Function

*c\_ltof* converts the four byte integer in *float register* to a float.

## Return Value

The resulting value is in *float register*. Flags have no meaningful value upon return.

## See Also

*c\_ftoi, c\_ftol, c\_itof, c\_itol*

# c\_ltor

## Description

Load memory into long register

## Syntax

```
; pointer to operand in x register  
    call c_ltor  
; result in float register
```

## Function

*c\_ltor* loads the four byte integer pointed by the **x** register into the *long register*.

## Return Value

The resulting value is in *long register*. Flags have no meaningful value upon return.

## See Also

*c\_rtol*

# c\_ludv

## Description

Quotient of unsigned long integer division

## Syntax

```
; left in long register  
; pointer to right in x register  
    call c_ludv  
; quotient in long register
```

## Function

*c\_ludv* divides the four byte integer in *long register* by the four byte integer pointered by the **x** register. Values are assumed to be unsigned. The dividend is returned if a division by zero is attempted.

## Return Value

The quotient is in *long register*. Flags are not meaningful upon return.

## See Also

*c\_ldiv*, *c\_lmod*, *c\_lumd*

# c\_lumd

## Description

Remainder of unsigned long integer division

## Syntax

```
; left in long register  
; pointer to right in x register  
    call c_lumd  
; remainder in long register
```

## Function

*c\_lumd* divides the four byte integer in *long register* by the four byte integer pointered by the **x** register. Values are assumed to be unsigned. The dividend is returned if a division by zero is attempted.

## Return Value

The remainder is in *long register*. Flags are not meaningful upon return.

## See Also

*c\_lmod*, *c\_ldiv*, *c\_ludv*

# c\_lursh

## Description

Unsigned long integer shift right

## Syntax

```
; operand in long register  
; shift count in a register  
    call c_lursh  
; result in long register
```

## Function

*c\_lursh* right shifts the four byte integer in *long register* by the number of bits specified by the **r** register. A zero count leaves the *long register* unchanged. No check is made for invalid counts. The value is assumed to be unsigned. The shift instruction used is therefore a logical shift.

## Return Value

The resulting value is in *long register*. Flags are not meaningful upon return.

## See Also

*c\_llsh*, *c\_lrsh*

# c\_lxor

## Description

Bitwise exclusive OR with long integers

## Syntax

```
; left in long integer  
; pointer to right in x register  
    call c_lxor  
; result in long register
```

## Function

*c\_lxor* operates a bitwise Exclusive OR between the contents of *long register* and the long pointed by the **x** register. Each operand is taken to be a four byte integer.

## Return Value

The result is in *long register*. The flags are not meaningful upon return.

## See Also

*c\_land*, *c\_lor*

# c\_lzmp

## Description

Compare a long integer to zero

## Syntax

```
; pointer to operand in x register  
    call c_lzmp  
; result in the flags
```

## Function

*c\_lzmp* compares the four byte integer pointed by the **x** register to zero.

## Return Value

Nothing, but the flags.

## See Also

*c\_lrzmp*



# c\_pgadc

## Description

Far pointer addition

## Syntax

```
; pointer to left in x register  
; right in a register  
    call c_pgadc  
; result in left
```

## Function

*c\_pgadc* performs the long addition of the unsigned char value in the **a** register and the far pointer pointed at by the **x** register.

## Return Value

The result is stored at the location in the **x** register. Flags are not meaningful upon return.

## See Also

*c\_lgadd*, *c\_lgadc*

# c\_pgadd

## Description

Far pointer addition

## Syntax

```
; pointer to left in x register  
; right in long register  
    call c_pgadd  
; result in left
```

## Function

*c\_pgadd* performs the long addition of the value in *long register* and the far pointer pointed at by the **x** register.

## Return Value

The result is stored at the location in the **x** register. Flags are not meaningful upon return.

## See Also

*c\_lgadd*, *c\_lgadc*

# c\_putlx

## Description

Put a long integer in memory

## Syntax

```
; long address in x  
; value in long register  
call c_putlx
```

## Function

*c\_putlx* puts the value in *long register* into memory using a pointer loaded in **x**.

## Return Value

None.

## See Also

*c\_getlx*, *c\_getwx*, *c\_putly*, *c\_putw*

# c\_putly

## Description

Put a long integer in memory

## Syntax

```
; long address in y  
; value in long register  
call c_putly
```

## Function

*c\_putly* puts the value in *long register* into memory using a pointer loaded in **y**.

## Return Value

None.

## See Also

*c\_getly*, *c\_getwy*, *c\_putlx*, *c\_putw*

# c\_putwf

## Description

Put a word in far memory

## Syntax

```
; word address in y and c_y  
; value in a and xl  
call c_putwf
```

## Function

*c\_putwf* puts the value in **a** and **xl** registers into far memory using a pointer loaded in **y** and **c\_y**.

## Return Value

None.

## See Also

*c\_getlx*, *c\_getly*, *c\_getw*, *c\_putlx*, *c\_putly*

# c\_pxtox

## Description

Get a far pointer from far memory

## Syntax

```
; far pointer address in x and c_x  
    call c_pxtox  
; result in x and c_x
```

## Function

*c\_pxtox* gets a far pointer from far memory using a pointer loaded in **x** and **c\_x**. The result is left in the **x** and **c\_x**.

## Return Value

The far pointer is loaded in the **x** register and the **c\_x** memory location. Flags have no meaningful value upon return.

## See Also

*c\_pxtoy*, *c\_pytox*, *c\_pytoy*

# c\_pxtoy

## Description

Get a far pointer from far memory

## Syntax

```
; far pointer address in x and c_x  
    call c_pxtoy  
; result in y and c_y
```

## Function

*c\_pxtoy* gets a far pointer from far memory using a pointer loaded in **x** and **c\_x**. The result is left in the **y** and **c\_y**.

## Return Value

The far pointer is loaded in the **y** register and the **c\_y** memory location. Flags have no meaningful value upon return.

## See Also

*c\_pxtox*, *c\_pytox*, *c\_pytoy*

# c\_pytox

## Description

Get a far pointer from far memory

## Syntax

```
; far pointer address in y and c_y  
    call c_pytox  
; result in x and c_x
```

## Function

*c\_pytox* gets a far pointer from far memory using a pointer loaded in **y** and **c\_y**. The result is left in the **x** and **c\_x**.

## Return Value

The far pointer is loaded in the **x** register and the **c\_x** memory location. Flags have no meaningful value upon return.

## See Also

*c\_pxtox*, *c\_pxtoy*, *c\_pytoy*



# c\_pytoy

## Description

Get a far pointer from far memory

## Syntax

```
; far pointer address in y and c_y  
    call c_pytoy  
; result in y and c_y
```

## Function

*c\_pytoy* gets a far pointer from far memory using a pointer loaded in **y** and **c\_y**. The result is left in the **y** and **c\_y**.

## Return Value

The far pointer is loaded in the **y** register and the **c\_y** memory location. Flags have no meaningful value upon return.

## See Also

*c\_pxtox*, *c\_pxtoy*, *c\_pytox*

# c\_rtofl

## Description

Store long register in far memory

## Syntax

```
; pointer to destination in x register and c_x  
; operand in long integer  
call c_rtofl
```

## Function

*c\_rtofl* store the four byte integer in *long register* into the memory location pointered by the **x** register and **c\_x**.

## Return Value

The resulting value is in the memory location pointered by **x** and **c\_x**.  
Flags have no meaningful value upon return.

## See Also

*c\_ftor*

# c\_rtol

## Description

Store long register in memory

## Syntax

```
; pointer to destination in x register  
; operand in long integer  
call c_rtol
```

## Function

*c\_rtol* store the four byte integer in *long register* into the memory location pointered by the **x** register.

## Return Value

The resulting value is in the memory location pointered by **x**. Flags have no meaningful value upon return.

## See Also

*c\_ltor*

# c\_sdivx

## Description

Quotient of signed char division

## Syntax

```
; dividend in x register  
; divisor in a register  
    call c_sdivx  
; quotient in x
```

## Function

*c\_sdivx* divides the signed integer in **x** by the signed byte in the **a** register. Values are assumed to be signed. If division by zero is attempted, the result is the unchanged dividend.

## Return Value

The quotient is in **x**. Flags are not meaningful upon return.

## See Also

*c\_cdivx*, *c\_cdivy*, *c\_sdivy*

# c\_sdivy

## Description

Quotient of signed char division

## Syntax

```
; dividend in y register  
; divisor in a register  
    call c_sdivy  
; quotient in y
```

## Function

*c\_sdivy* divides the signed integer in **y** by the signed byte in the **a** register. Values are assumed to be signed. If division by zero is attempted, the result is the unchanged dividend.

## Return Value

The quotient is in **y**. Flags are not meaningful upon return.

## See Also

*c\_cdivx*, *c\_cdivy*, *c\_sdivy*

# c\_smodx

## Description

Remainder of signed char division

## Syntax

```
; dividend in x register  
; divisor in a register  
    call c_smodx  
; remainder in x
```

## Function

*c\_smodx* divides the signed integer in **x** by the signed byte in the **a** register. Values are assumed to be signed. If division by zero is attempted, the result is the unchanged dividend.

## Return Value

The remainder is in **x**. Flags are not meaningful upon return.

## See Also

*c\_smody*

# c\_smody

## Description

Remainder of signed char division

## Syntax

```
; dividend in y register  
; divisor in a register  
    call c_smody  
; remainder in y
```

## Function

*c\_smody* divides the signed integer in **y** by the signed byte in the **a** register. Values are assumed to be signed. If division by zero is attempted, the result is the unchanged dividend.

## Return Value

The remainder is in **y**. Flags are not meaningful upon return.

## See Also

*c\_smodx*

# c\_smul

## Description

Multiply long integer by unsigned byte

## Syntax

```
; left in long register  
; right byte in a register  
    call c_smul  
; result in long register
```

## Function

*c\_smul* multiplies the four byte integer in *long register* by the unsigned byte in the **a** register. No check is made for overflow.

## Return Value

The resulting value is in *long register*. Flags are not meaningful upon return.

## See Also

*c\_lgmul*



# c\_uitof

## Description

Convert unsigned integer into float

## Syntax

```
; operand in x  
    call c_uitof  
; result in float register
```

## Function

*c\_uitof* converts the two byte unsigned integer in the **x** register to a float stored in *float register*.

## Return Value

The resulting value is in *float register*. Flags have no meaningful value upon return.

## See Also

*c\_itof*, *c\_ltof*, *c\_ultof*, *c\_xtof*, *c\_uxtof*

# c\_uitol

## Description

Convert unsigned integer into long

## Syntax

```
; operand in a and xl  
    call c_uitol  
; result in long register
```

## Function

*c\_uitol* converts the two byte unsigned integer in the **a** and **xl** register pair, to a long integer stored in *long register*.

## Return Value

The resulting value is in *long register*. Flags have no meaningful value upon return.

## See Also

*c\_itol*, *c\_xtol*, *c\_uxtol*

# c\_uitolx

## Description

Convert unsigned integer into long

## Syntax

```
; operand in x
    call c_uitolx
; result in long register
```

## Function

`c_uitolx` converts the two byte unsigned integer in the **x** register, to a long integer stored in *long register*.

## Return Value

The resulting value is in *long register*. Flags have no meaningful value upon return.

## See Also

`c_itol`, `c_xtol`, `c_uxtol`

# c\_uitoly

## Description

Convert unsigned integer into long

## Syntax

```
; operand in y  
    call c_uitoly  
; result in long register
```

## Function

*c\_uitoly* converts the two byte unsigned integer in the **y** register, to a long integer stored in *long register*.

## Return Value

The resulting value is in *long register*. Flags have no meaningful value upon return.

## See Also

*c\_itol*, *c\_xtol*, *c\_uxtol*

# c\_ultof

## Description

Convert unsigned long integer into float

## Syntax

```
; long in long register  
    call c_ultof  
; result in float register
```

## Function

*c\_ultof* converts the four unsigned byte integer in *long register* to a float.

## Return Value

The resulting value is in *float register*. Flags have no meaningful value upon return.

## See Also

*c\_itoi*, *c\_ltof*, *c\_xtof*, *c\_uitof*, *c\_uxtof*

# c\_umul

## Description

Multiply unsigned integers with long result

## Syntax

```
; left in x register  
; right in y register  
    call c_umul  
; result in long register
```

## Function

`c_umul` multiplies the two byte unsigned integer in the **x** register by the two byte unsigned integer in the **y** register.

## Return Value

The resulting value is in *long register*. Flags are not meaningful upon return.

## See Also

`c_vmul`

# `c_vmul`

## Description

Multiply signed integers with long result

## Syntax

```
; left in x register  
; right in y register  
    call c_vmul  
; result in long register
```

## Function

`c_vmul` multiplies the two byte signed integer in the **x** register by the two byte signed integer in the **y** register.

## Return Value

The resulting value is in *long register*. Flags are not meaningful upon return.

## See Also

`c_umul`

# c\_xtopy

## Description

Store a far pointer into far memory

## Syntax

```
; value in x and c_x  
; far pointer address in y and c_y  
call c_xtopy
```

## Function

*c\_xtopy* stores the far pointer in **x** and **c\_x** into far memory using a pointer loaded in **y** and **c\_y**.

## Return Value

Flags have no meaningful value upon return.

## See Also

*c\_ytopx*



# c\_xymov

## Description

Copy a structure into another

## Syntax

```
; pointer to source in y  
; pointer to destination in x  
; size in a register  
call c_xymov
```

## Function

*c\_xymov* copy the source structure pointed by the memory location **y** into the structure pointed by the **x** register. The structure size is located in the **a** register.

## Return Value

None.

## See Also

*c\_xymov*, *c\_yxmov*, *c\_xymvx*, *c\_yxmvx*

# c\_xymvf

## Description

Copy a structure in far memory

## Syntax

```
; pointer to source in y and c_y  
; pointer to destination in x and c_x  
; size in a  
call c_xymvf
```

## Function

*c\_xymvf* copy the source structure pointed by the **y** register and the memory location pointed by **c\_y** into the structure pointed by the **x** register and the memory location pointed by **c\_x**. The structure size is in the **a** register.

## Return Value

None.

## See Also

*c\_xymov*, *c\_yxmov*, *c\_xymvy*

# c\_xymvfl

## Description

Copy a large structure in far memory

## Syntax

```
; pointer to source in y and c_y  
; pointer to destination in c_x  
; size in x  
    call c_xymvfl
```

## Function

*c\_xymvfl* copy the source structure pointed by the **y** register and the memory location pointed by **c\_y** into the structure pointed by the memory location pointed by **c\_x**. The structure size is in the **x** register.

## Return Value

None.

## See Also

*c\_xymov*, *c\_yxmov*, *c\_xymvy*

# c\_xymvx

## Description

Copy a structure into another

## Syntax

```
; pointer to source in y  
; pointer to destination in x  
; size in a register  
call c_xymvx
```

## Function

*c\_xymvx* copy the source structure pointed by the memory location **y** into the structure pointed by the **x** register. The structure size is located in the **a** register.

## Return Value

None.

## See Also

*c\_xymov*, *c\_yxmov*, *c\_xymvx*, *c\_yxmvx*

# c\_xymvxl

## Description

Copy a large structure into another

## Syntax

```
; pointer to source in y  
; pointer to destination in c_x  
; size in x register  
call c_xymvxl
```

## Function

*c\_xymvxl* copy the source structure pointed by the memory location **y** into the structure pointed by the **c\_x** memory location. The structure size is located in the **x** register.

## Return Value

None.

## See Also

*c\_xymov*, *c\_yxmov*, *c\_xymvx*, *c\_yxmvx*

# c\_ytopx

## Description

Store a far pointer into far memory

## Syntax

```
; value in y and c_y  
; far pointer address in x and c_x  
call c_ytopx
```

## Function

*c\_ytopx* stores the far pointer in **y** and **c\_y** into far memory using a pointer loaded in **x** and **c\_x**.

## Return Value

Flags have no meaningful value upon return.

## See Also

*c\_xtopy*

# **c\_yxmov**

## **Description**

Copy a structure into another

## **Syntax**

```
; pointer to source in x  
; pointer to destination in y  
; size in a  
call c_yxmov
```

## **Function**

*c\_yxmov* copy the source structure pointed by the **x** register into the structure pointed by the **y** register. The structure size is in the **a** register.

## **Return Value**

None.

## **See Also**

*c\_xymov*, *c\_xymvx*, *c\_yxmvx*

# c\_yxmvmf

## Description

Copy a structure in far memory

## Syntax

```
; pointer to source in x and c_x  
; pointer to destination in y and c_y  
; size in a  
call c_yxmvmf
```

## Function

*c\_yxmvmf* copy the source structure pointed by the **x** register and the memory location pointed by **c\_x** into the structure pointed by the **y** register and the memory location pointed by **c\_y**. The structure size is in the **a** register.

## Return Value

None.

## See Also

*c\_xymov*, *c\_yxmvmov*, *c\_xymvmy*



# c\_yxmvfl

## Description

Copy a large structure in far memory

## Syntax

```
; pointer to source c_x  
; pointer to destination in y and c_y  
; size in x  
call c_yxmvfl
```

## Function

*c\_yxmvfl* copy the source structure pointed by the memory location pointed by **c\_x** into the structure pointed by the **y** register and the memory location pointed by **c\_y**. The structure size is in the **x** register.

## Return Value

None.

## See Also

*c\_xy mov*, *c\_yx mov*, *c\_xy mv y*

# c\_yxmvx

## Description

Copy a structure into another

## Syntax

```
; pointer to source in x  
; pointer to destination in y  
; size in a  
call c_yxmvx
```

## Function

*c\_yxmvx* copy the source structure pointed by the **x** register into the structure pointed by the **y** register. The structure size is in the **a** register.

## Return Value

None.

## See Also

*c\_xyymov*, *c\_xyymvx*, *c\_yxmvx*

# `c_yxmvxl`

## Description

Copy a large structure into another

## Syntax

```
; pointer to source in c_x  
; pointer to destination in y  
; size in x  
call c_yxmvxl
```

## Function

`c_yxmvxl` copy the source structure pointed by the `c_x` memory location into the structure pointed by the `y` register. The structure size is in the `x` register.

## Return Value

None.

## See Also

`c_xymov`, `c_xymvx`, `c_yxmvx`



# Compiler Passes

---

The information contained in this appendix is of interest to those users who want to modify the default operation of the cross compiler by changing the configuration file that the **cxstm8** compiler uses to control the compilation process.

This appendix describes each of the passes of the compiler:

<b>cpstm8</b>	the parser
<b>cgstm8</b>	the code generator
<b>costm8</b>	the assembly language optimizer

## The *cpstm8* Parser

*cpstm8* is the parser used by the C compiler to expand *#defines*, *#includes*, and other directives signalled by a *#*, parse the resulting text, and outputs a sequential file of flow graphs and parse trees suitable for input to the code generator *cgstm8*.

### Command Line Options

*cpstm8* accepts the following options, each of which is described in detail below:

```
cpstm8 [options] file
    -ad      expand defines in assembly
    -c99     c99 type behaviour
    -cc      do not cast const expressions
    -ck      extra type checkings
    -cp      no constant propagation
    -csb     check signed bitfields
    -d*>     define symbol=value
    -e       run preprocessor only
    +e*      error file name
    -ge      eclipse error messages
    -h*>     include header
    -i*>     include path
    -ku      keep unused static
    -l       output line information
    -md      make dependencies
    -m#      model configuration
    -nb      no bitfield packing
    -nc      no const replacement
    -ne      no enum optimization
    -np      allow pointer narrowing
    -ns      do not share locals
    -o*      output file name
    -p       need prototypes
    -pb      pack bit variables
    -rb      reverse bitfield order
    -s       do not reorder locals
    -sa      strict ANSI conformance
    -u       plain char is unsigned
    -w#      enable warnings
    -xd      debug info for data
    -xf      full path in debug info
    -xp      no path in debug info
    -xu      no debug info if unused
    -xx      extended debug info
    -x       output debug info
```

## Parser Option Usage

Option	Description
<b>-ad</b>	enable <code>#define</code> expansion inside inline assembly code between <code>#asm</code> and <code>#endasm</code> directives. By default, <code>#define</code> symbols are expanded only in the C code.
<b>-c99</b>	authorize the repetition of the <code>const</code> and <code>volatile</code> modifiers in the declaration either directly or indirectly in the typedef.
<b>-cc</b>	do not apply standard type casting to the result of a constant expression. This option allows compatibility with parsers previous to version V4.5p. These previous parsers were behaving as if all constants were considered of type <code>long</code> instead of the default type <code>int</code> . Such expressions were allowing intermediate results to become larger than an <code>int</code> without any truncation.
<b>-ck</b>	enable extra type checking. For more information, see " <a href="#">Extra verifications</a> " below.
<b>-cp</b>	disable the constant propagation optimization. By default, when a variable is assigned with a constant, any subsequent access to that variable is replaced by the constant itself until the variable is modified or a flow break is encountered (function call, loop, label ...).
<b>-csb</b>	produce an error message if a bitfield is declared explicitly with the <b>signed</b> keyword. By default, the compiler silently ignores the signed feature and handles all bitfields as unsigned values.
<b>-d*^</b>	specify <code>*</code> as the name of a user-defined preprocessor symbol ( <b>#define</b> ). The form of the definition is <b>-dsymbol[=value]</b> ; the symbol is set to <code>1</code> if value is omitted. You can specify up to 128 such definitions.
<b>-e</b>	run preprocessor only. <i>cpstm8</i> only outputs lines of text.
<b>+e*</b>	log errors in the text file <code>*</code> instead of displaying the messages on the terminal screen.
<b>-ge</b>	produce error messages directly compatible with the Eclipse environment.



## Parser Option Usage (cont.)

Option	Description						
<b>-h*&gt;</b>	include files before to start the compiler process. You can specify up to 128 files.						
<b>-i*&gt;</b>	specify include path. You can specify up to 128 different paths. Each path is a directory name, <b>not</b> terminated by any directory separator character, or a file containing an unlimited list of directory names.						
<b>-ku</b>	keep unused statics. By default, unused statics are removed.						
<b>-l</b>	output line number information for listing or debug.						
<b>-md</b>	create only a list of 'make' compatible dependencies consisting for each source file in the object name followed by a list of header files needed to compile that file.						
<b>-m#</b>	the value # is used to configure the parser behaviour. It is a two bytes value, the upper byte specifies the default space for variables, and the lower byte specifies the default space for functions. A space byte is the or'ed value between a size specifier and several optional other specifiers. The allowed size specifiers are: <div data-bbox="551 922 786 1043" data-label="Table"> <table> <tr> <td><b>0x10</b></td><td>@tiny</td></tr> <tr> <td><b>0x20</b></td><td>@near</td></tr> <tr> <td><b>0x30</b></td><td>@far</td></tr> </table> </div>	<b>0x10</b>	@tiny	<b>0x20</b>	@near	<b>0x30</b>	@far
<b>0x10</b>	@tiny						
<b>0x20</b>	@near						
<b>0x30</b>	@far						
	Allowed optional specifiers are: <div data-bbox="555 1131 790 1212" data-label="Table"> <table> <tr> <td><b>0x01</b></td><td>@pack</td></tr> <tr> <td><b>0x04</b></td><td>@nostack</td></tr> </table> </div>	<b>0x01</b>	@pack	<b>0x04</b>	@nostack		
<b>0x01</b>	@pack						
<b>0x04</b>	@nostack						
<b>-nb</b>	do not pack bitfields. By default, trailing unused bits in the last bitfield of a structure are removed if this saves at least one byte.						

**Parser Option Usage (cont.)**

Option	Description
<b>-nc</b>	do not replace an access to an initialized const object by its value. By default, the usage of a const object whose value is known is replaced by its constant value.
<b>-ne</b>	do not optimize size of <i>enum</i> variables. By default, the compiler selects the smallest integer type by checking the range of the declared <i>enum</i> members. This mechanism does not allow incomplete <i>enum</i> declaration. When the <b>-ne</b> option is selected, all <i>enum</i> variables are allocated as <i>int</i> variables, thus allowing incomplete declarations, as the knowledge of all the members is no more necessary to choose the proper integer type.
<b>-np</b>	allow pointer narrowing. By default, the compiler refuses to cast the pointer into any smaller object. This option should be used carefully as such conversions are truncating addresses.
<b>-ns</b>	do not share independent local variables. By default, the compiler tries to overlay variables in the same memory location or register if they are not used concurrently.
<b>-o*</b>	write the output to the file *. Default is STDOUT for output if <b>-e</b> is specified. Otherwise, an output file name is required.
<b>-p</b>	enforce prototype declaration for functions. An error message is issued if a function is used and no prototype declaration is found for it. By default, the compiler accepts both syntaxes without any error.
<b>-pb</b>	pack <b>_Bool</b> local variables. By default, <b>_Bool</b> local variables are allocated on one byte each.
<b>-rb</b>	reverse the bitfield fill order. By default, bitfields are filled from <b>less</b> significant bit ( <b>LSB</b> ) to <b>most</b> significant bit ( <b>MSB</b> ). If this option is specified, filling works from most significant bit to less significant bit.

## Parser Option Usage (cont.)

Option	Description
<b>-s</b>	do not reorder local variables. By default, the compiler sorts the local variables of a function in order to allocate the most used variables as close as possible to the frame pointer. This allows to use the shortest addressing modes for the most used variables.
<b>-sa</b>	enforce a strict ANSI checking by rejecting any syntax or semantic extension. This option also disables the enum size optimization ( <b>-ne</b> ).
<b>-u</b>	take a plain char to be of type <b>unsigned char</b> , not signed char. This also affects in the same way strings constants.
<b>-w#</b>	enable warnings from level #. By default, warnings are disabled. For more information, see " <a href="#">Warning Levels</a> " below.
<b>-x</b>	generate debugging information for use by the cross debugger or some other debugger or in-circuit emulator. The default is to generate no debugging information.
<b>-xd</b>	add debug information in the object file only for data objects, hiding any function.
<b>-xf</b>	prefix filenames in the debug information with absolute full path name.
<b>-xp</b>	do not prefix filenames in the debug information with any absolute path name. Debuggers will have to be informed about the actual files location.
<b>-xu</b>	do not produce debug information for localized variables if they are not used. By default, the compiler produces a complete debug information regardless the variable is accessed or not.
<b>-xx</b>	add debug information in the object file for any label defining code or data.

## Warning Levels

The option enabling warnings also allows to define a minimum level which reduces the amount of warning produced. They are grouped on 7 levels depending on their effects on the resulting code from the smallest (1) to the highest (7). The **+warn** compiler option activates warning

from level 1. To activate warnings from a different level, the **-pw#** option can be used, where # is a starting level.

The list of warning messages is shared with the extra verifications ones described in the next paragraph along with their associated level displayed between parenthesis.

## Extra verifications

This paragraph describes the checkings done by the **-ck** parser option (**+strict** compiler option) or when warnings are enabled (**+warn** or **-pw#** compiler option) according to the error message produced.

**compare out of range** (level 2) - a comparison is made with a constant larger (or smaller) than the possible values for the type of the compared expression.

**constant assignment in a test** (level 4) - a constant is assigned to a variable in a test expression.

**float value too large for integer cast** (level 1) - a float constant is cast to an integer or a long but is larger than the maximum value of the cast type.

**implicit int type in struct/union declaration** (level 5)

**implicit int type in global declaration** (level 5)

**implicit int type in local declaration** (level 5)

**implicit int type in argument declaration** (level 5) - an object is declared without an explicit type and is defaulted to **int** according to the ANSI standard.

**missing explicit return** (level 6) - a function is not ending with a return statement.

**missing prototype** (level 2) - a function has been called without any previous prototype. The **-pp** option also produces this message.

**missing return expression** (level 6) - a return statement without expression is specified in a function with a non void return type.

**shift count out of range** (level 2) - a shift count is larger than the bit size of the shifted expression.

**truncating assignment** (level 1) - an expression is assigned to a variable and has a type larger than the variable one.

**truncating constant cast** (level 1) - a constant is cast to a smaller type, which truncates its initial value.

**unreachable code** (level 4) - a code sequence cannot be reached due to previous optimizations.

**value out of range** (level 2) - a constant is assigned to a variable and is larger (or smaller) than the possible set of values for that type.

**<variable> used before set** (level 7) - a local variable has been used before being initialized by an explicit assignment.

## Return Status

*cpstm8* returns success if it produces no error diagnostics.

## Example

*cpstm8* is usually invoked before *cgstm8* the code generator, as in:

```
cpstm8 -o \2.cx1 -u -i \cosmic\hstm8 file.c
cgstm8 -o \2.cx2 \2.cx1
```

## The *cgstm8* Code Generator

***cgstm8*** is the code generating pass of the C compiler. It accepts a sequential file of flow graphs and parse trees from *cpstm8* and outputs a sequential file of assembly language statements.

As much as possible, the compiler generates freestanding code, but, for those operations which cannot be done compactly, it generates inline calls to a set of machine-dependent runtime library routines.

### Command Line Options

*cgstm8* accepts the following options, each of which is described in detail below:

```
cgstm8 [options] file
-a      optimize _asm code
-bss    do not use bss
-ck     check stack frame
-ct     constants in code
-dl#    output line information
-dp     no divide protection
+e*     error file name
-f      full source display
-fl     far library calls
-ge     eclipse error messages
-i      inline long machine calls
-l      output listing
-na     do not xdef alias name
-nc     functions do not cross section
-no     do not use optimizer
-o*     output file name
-sf     split function sections
-v      verbose
```

### Code generator Option Usage

Option	Description
<b>-a</b>	optimize <i>_asm</i> code. By default, the assembly code inserted by a <i>_asm</i> call is left unchanged by the optimizer.
<b>-bss</b>	inhibit generating code into the <i>bss</i> section.

## Code generator Option Usage (cont.)

Option	Description
<b>-ck</b>	enable stack overflow checking.
<b>-ct</b>	output constant in the <b>.text</b> section. By default, the compiler outputs literals and constants in the <b>.const</b> section.
<b>-dl#</b>	produce line number information. # must be either '1' or '2'. Line number information can be produced in two ways: 1) function name and line number is obtained by specifying <b>-dl1</b> ; 2) file name and line number is obtained by specifying <b>-dl2</b> . All information is coded in symbols that are in the debug symbol table.
<b>-dp</b>	do not produce any protection sequence on interrupt functions using divide instructions.
<b>+e*</b>	log errors in the text file * instead of displaying the messages on the terminal screen.
<b>-f</b>	merge all C source lines of functions producing code into the C and Assembly listing. By default, only C lines actually producing assembly code are shown in the listing.
<b>-fl</b>	use <i>callf</i> instruction for machine library calls, used for models allowing large applications. By default, machine library functions are called with a <i>call</i> instruction allowing only 64K application. This option is configured by the memory model selection.
<b>-ge</b>	produce error messages directly compatible with the Eclipse environment.
<b>-i</b>	produce faster code by inlining machine library calls for long integers handling. The code produced will be larger than without this option.
<b>-l</b>	merge C source listing with assembly language code; listing output defaults to <i>&lt;file&gt;.ls</i> .
<b>-na</b>	do not produce an <i>xdef</i> directive for the <i>equate</i> names created for each C object declared with an absolute address.
<b>-nc</b>	do not allow functions to cross a section boundary.
<b>-no</b>	do not produce special directives for the post-optimizer.

**Code generator Option Usage (cont.)**

Option	Description
<b>-o*</b>	write the output to the file * and write error messages to STDOUT. The default is STDOUT for output and STDERR for error messages.
<b>-sf</b>	produce each function in a different section, thus allowing the linker to suppress a function if it is not used by the application. By default, all the functions are packed in a single section.
<b>-v</b>	When this option is set, each function name is send to STDERR when <i>cgstm8</i> starts processing it.

**Return Status**

*cgstm8* returns success if it produces no diagnostics.

**Example**

*cgstm8* usually follows *cpstm8* as follows:

```
cpstm8 -o \2.cx1 -u -i\cosmic\hstm8 file.c
cgstm8 -o \2.cx2 \2.cx1
```



## The *costm8* Assembly Language Optimizer

***costm8*** is the code optimizing pass of the C compiler. It reads source files of *STM8* assembly language source code, as generated by the *cgstm8* code generator, and writes assembly language statements. *costm8* is a peephole optimizer; it works by checking lines function by function for specific patterns. If the patterns are present, *costm8* replaces the lines where the patterns occur with an optimized line or set of lines. It repeatedly checks replaced patterns for further optimizations until no more are possible. It deals with redundant load/store operations, constants, stack handling, and other operations.

### Command Line Options

*costm8* accepts the following options, each of which is described in detail below:

```
costm8 [options] <file>
  -c      keep original lines as comments
  -d*     disable specific optimizations
  -f#     minimum code factorization
  -o*     output file name
  -v      print efficiency statistics
```

### Optimizer Option Usage

Option	Description
<b>-c</b>	leave removed instructions as comments in the output file.
<b>-d*</b>	specify a list of codes allowing specific optimizations functions to be selectively disabled.
<b>-f#</b>	define the minimum of bytes for activating the code factorization. Any value smaller than 4 disables the feature. The default value is 7.
<b>-o*</b>	write the output to the file * and write error messages to STDOUT. The default is STDOUT for output and STDERR for error messages.
<b>-v</b>	write a log of modifications to STDERR. This displays the number of removed instructions followed by the number of modified instructions.

If *<file>* is present, it is used as the input file instead of the default STDIN.

## Disabling Optimization

When using the optimizer with the **-c** option, lines which are changed or removed are kept in the assembly source as comment, followed by a code composed with a letter and a digit, identifying the internal function which performs the optimization. If an optimization appears to do something wrong, it is possible to disable selectively that function by specifying its code with the **-d** option. Several functions can be disabled by specifying a list of codes without any whitespaces. The code letter can be enter both lower or uppercase.

## Return Status

*costm8* returns success if it produces no diagnostics.

## Example

*costm8* is usually invoked after *cgstm8* as follows:

```
cpstm8 -o \2.cx1 -u -i\cosmic\hstm8 file.c
cgstm8 -o \2.cx2 \2.cx1
costm8 -o file.s \2.cx2
```

# *Index*

## **Symbols**

- #asm
  - directive 466
- #asm directive 55
- #endasm
  - directive 466
- #endasm directive 55
- #pragma
  - asm directive 55
  - directive for inlining 55
  - endasm directive 55
  - space directive 49
- +dep
  - linker dependency directive 276
- +grp directive 271
- +modsl memory model 17
- +seg option 267
- .bsct section 183
- .const
  - output section 473
  - segment 278
- @eeprom
  - type qualifier 12, 51
- @far
  - .fconst section 41
  - .fdata section 41, 53
  - function 41, 63
  - modifier 39, 63
  - pointer,size 39
- @inline
  - functions 61
  - modifier 61
- @interrupt
  - function 59
  - qualifier 59
- @near
  - .data,.bss sections 53
  - function 41
  - modifier 50, 63
  - modifier,mods 39
  - modifier,mods0 39
  - pointer,size 39
  - variable 63
- @near type qualifier 25
- @noprđ qualifier 60
- @nosvf qualifier 59
- @svlreg qualifier 59
- @tiny
  - .bsct,.ubset sections 53
  - modifier 49
  - modifier,modsl 40
  - modifier,modsl0 39
  - space modifier 63
  - variable 63
- @vector modifier 60
- \_\_ckdesc\_\_ 1 287
- \_\_idesc\_\_ 283, 284
- \_\_asm
  - argument string length 57
  - argument string size 57
  - assembly sequence 56
  - code optimization 472
  - in expression 57
  - lowercase mnemonics 57

---

- return type 57
- uppercase mnemonics 57
- `_asm()`
  - function 86
  - inserting assembler function 82
- `_Bool`
  - assign expression to 42
  - consecutive fields 42
  - data 63
  - pack local variable 468
  - packed variables 42
  - referencing absolute address 46
  - type name 42
  - variable 42
- `_checksum` function 98
- `_checksum16` function 100
- `_checksum16x` function 101
- `_checksumx` function 99
- `_fctcpy` function 111
- `_sbreak` function 66

## Numerics

8-bit precision,operation 11

## A

- abort function 87
- abs function 88
- absolute
  - address 302
  - address in listing 314
  - full path name 469
  - hex file generator 9
  - listing file 314
  - listing utility 10
  - map section 178
  - path name 469
  - reference address 46
  - section relocation 277
  - symbol 270
  - symbol in library 317
  - symbol table 265
  - symbol tables 294

- symbol,flagged 294
- absolute section 237, 247
- acos function 89
- address
  - default format 311, 315
  - logical end 269
  - logical start segment 277
  - logical start set 269
  - paged format 311, 315
  - physical 269
  - physical end 267
  - physical start 267
  - physical start segment 277
  - set logical 269
- align directive 196
- allocate memory block 201
- allocate storage for constants 200
- application
  - embedded 260
  - non-banked 312
  - system bootstrap 260
- Arccosine 89
- Arcsine 90
- Arctangent 91
- Arctangent of  $y/x$  92
- argument
  - formatted output to buffer 154
  - formatted output to stdout 144
- asin function 90
- assembler
  - branch shortening 194
  - C style directives 195
  - code inline 56
  - conditional branch range 194
  - conditional directive 190
  - create listing file 179
  - debug information
    - add line 180
    - label 180
  - endm directive 187
  - expression 186
  - filling byte 179, 196

---

- include directive 193
- label 183
- listing process 314
- listing stream 181
- macro
  - instruction 187
- macro argument 188
- macro directive 187
- macro parameter 188
- old syntax 194
- operator set 186
- section name 191
- section predefinition 192
- sections 191
- special parameter \# 189
- special parameter \\* 190
- special parameter \0 189
- switch directive 191
- xbit directive 254
- assembly language
  - code optimizer 475
- atan function 91
- atan2 function 92
- atof function 93
- atoi function 94
- atol function 95

## B

- bank
  - automatic segment creation 270
  - default mode 325
  - size setting 267
  - switched system 277
- base directive 197
- bias
  - segment parameter 277
  - setting 278
- bit
  - address 192
  - address value 278
  - allocated section 193
  - attribute section 193

- define aliases 274
  - number 278
  - segment 278
- bit segment
  - initialization 279
- bitfield
  - compiler reverse option 79
  - default order 468
  - filling 468
  - filling order 79
  - reverse order 468
  - sign check 466
- bootloader 284
- boundary
  - round up 269
- bsct directive 198
- buffer
  - convert to double 93, 170
  - convert to integer 94
  - convert to long 95, 171
  - convert to unsigned long 172
  - copy from one to another 138, 139
  - copy to eeprom 105

## C

- C interface
  - extra character for far function 63
  - underscore character prefix 63
- C interface to assembly language 63
- C library
  - floating point functions 83
  - integer functions 82
  - macro functions 83
  - package 82
- C source
  - lines merging 473
- c\_ prefix 357
- c\_lreg
  - memory byte 59
- c\_x
  - memory byte 59
- c\_y

---

- memory byte 59
- call
  - instruction 473
- callf
  - instruction 473
- carry function 96
- casting 466
- ceil function 97
- char
  - signed 469
  - unsigned 469
- character
  - fill throughout eeprom buffer 107
  - underscore,start 68
- checksum
  - ck option 287
  - crc 288
  - functions 287
  - ik option 288
- clabs utility 314
- clib utility 317
- clist directive 199, 214, 216, 217, 218, 219, 220, 221, 222, 223, 224
- clst utility 306
- cobj utility 320
- code
  - factorization 78, 475
  - smaller 78
- code generator
  - compiler pass 472
  - error log file 473
- code optimizer
  - compiler pass 475
- code/data, no output 267
- compare for lexical order 164
- compiler
  - ANSI checking 469
  - assembler 9
  - assembler option specification 76
  - C preprocessor and language parser 8
  - code generation option specification 77
  - code generator 9
  - code optimization 10
  - code optimizer 9
  - combination of options 354
  - command line option 74
  - configuration file 352
  - configuration file specification 76
  - configuration file,predefined option 78
  - create assembler file only 77
  - debug information,produce 78
  - default behavior 74
  - default configuration file 76
  - default file names 80
  - default operations 463
  - default options 74, 352
  - driver 4
  - error files path specification 76
  - error message 74
  - exclusive options 354
  - flags 6
  - generate error 327
  - generate error file 81
  - generate listing 81
  - header files 84
  - include path definition 77
  - invoke 74
  - listing file 77
  - listing file path specification 76
  - log error file 76
  - name 74
  - object file path specification 76
  - optimizer option specification 77
  - options 74
  - options request 74
  - parser option specification 77
  - predefined option selection 78
  - preprocessed file only 77
  - produce assembly file 19
  - produce listing file 19
  - programmable option 352, 354
  - specific options 4

- 
- specify options 75
  - stack long model option 78
  - stack short model option 78
  - temporary files path 77
  - type checking 79, 466
  - user-defined preprocessor symbol 76
  - verbose mode 20, 77
- compute 173
- const
- @near memory space 44
  - data 43
  - qualifier 43
- constant
- in .text section 473
  - numeric 185
  - prefix character 185
  - string 185
  - string character 186
  - suffix character 185
- convert
- ELF/DWARF format 324
  - hex format 310
- copy 150
- cos function 102
- cosh function 103
- cprd utility 304
- cross-reference
- information 179
  - table in listing 182
- ctat utility 322
- cvdwarf utility 324
- D**
- d\_prefix 357
- data
- @far modifier 21
  - @far pointer representation 70
  - @near modifier 21
  - @near pointer representation 70
  - @tiny modifier 21
  - @tiny pointer representation 70
  - automatic initialization 36
  - char representation 70
  - float representation 70
  - initialized 48
  - initialization 25
  - int representation 70
  - long int representation 70
  - short int representation 70
- data object
- automatic 304
  - scope 302
  - type 302
- dc directive 200
- dcb directive 201
- debug information
- adding 469
- debug symbol
- build table 289
  - in object file 181
  - table 302
- debugging
- data 302
  - support tools 301
- debugging information
- data object 302
  - extract 304
  - generate 302, 469
  - line number 302
  - print file 304
  - print function 304
- default base for numerical constants 197
- default placement
- .bit segment 278
  - .bsct segment 278
  - .bss segment 278
  - .data segment 278
  - .text segment 278
- definition 289
- DEFs 289
- dependency
- between function 276
- descriptor
- host to 268
- div function 104
- dlist directive 202

---

ds directive 203

## E

Eclipse

- error messages 76, 179, 265, 466, 473

eeepcpy function 105

eepera function 106

Eeprom

- STM8L library 281

eeeprom

- @near modifier 51

- erase 106

- location 12, 51

eeeprom support

- STM8L 52

eeepset function 107

ELF/DWARF

- format converter 10

else directive 204, 205, 208, 214, 216, 222

end directive 206

end5 directive 210

endc directive 216, 222

endif directive 204, 207, 208, 214

endm directive 209, 229, 232, 244

endr 240, 241

enum

- size optimization 468

environment symbol 193

equ directive 211, 249

error

- assembler log file 179

- file name 81

- log file 265

- message 10

- message list 327

- multiply defined symbol 184, 293

- undefined symbol 290

- undefined symbol in listing 180

error message 213

even directive 212

executable image 310

exit 108

exp function 109

expression

- evaluation 187

- high 187

- low 187

- page 187

## F

fabs function 110

fail directive 213

faster code

- production 473

file length restriction 302

filling byte 203, 212, 237

float

- single precision library 280

floating point library

- link 82

Floating Point Library Functions 83

floor function 112

fmod function 113

format

- ELF/DWARF 324

frexp function 114

function

- @inline modifier 61

- arguments 304

- enforce prototype declaration 79, 468

- in separate section 79

- prototype declaration 79, 468

- recursive 295

- returning int 85

- suppress 474

- suppress unused 79

function arguments 304

Functions Implemented as Macros 83

## G

generate



---

- .bss section 63
  - hex record 269
  - in .bit section 63
  - in .bss section 63
  - in .const section 63
  - in .data section 63
  - in .fconst section 63
  - in .fdata section 63
  - in .text section 63
  - in .ubss section 63
  - listing file 181
  - object file 181
- getchar function 115
- gets function 116
- group
  - option 262

## H

- heap
  - allocation 66
  - location 68
  - name 66
  - pointer 66
  - size 66
  - stack 66
  - start 66
  - top 66
- help option 7

## I

- IEEE
  - Floating Point Standard 70
- if directive 204, 208, 214
- if directive 207
- ifc directive 215
- ifdef directive 216
- ifeq directive 217
- ifge directive 218
- ifgt directive 219
- ifle directive 220
- iflt directive 221
- ifnc directive 224

- ifndef directive 222
- ifne directive 223
- imask function 117
- include
  - directory names list 77, 179, 467
  - file 272
  - file before 467
  - module 280
  - object file 272
  - path specification 467
  - specify path 467
- include directive 225
- initialization
  - automatic 283
  - define option 268
  - descriptor 283
  - descriptor address 284
  - descriptor format 283
  - first segment 283
  - initialized segments 283
  - marker 268
  - startup routine 284
- initialize storage for constants 200
- inline
  - @usea modifier 62
  - @usex modifier 62
  - assembly code 55, 56
  - block inside a function 55
  - block outside a function 55
  - carry function 61
  - function 61
  - header function 84
  - imask function 61
  - irq function 61
  - machine library calls 78, 473
  - produce faster code 78
  - user macro name 62
  - with \_asm function 56, 57
  - with pragma sequences 55
- input and output 45
- input to output 150
- input/output 46

---

- integer
  - library 280
- interrupt
  - function 60
  - function in map 295
  - handler 59
  - hardware 59
  - software 59
  - vectors table 60
- irq function 118
- isalnum function 119
- isalpha function 120
- isctrl function 121
- isdigit function 122
- isgraph function 123
- islower function 124
- isprint function 125
- ispunct function 126
- isspace function 127
- isupper function 128
- isxdigit function 129

## L

- labs function 130
- ldexp function 131
- ldiv function 132
- library
  - build and maintain 10
  - building and maintaining 317
  - create 317
  - delete file 317
  - extract file 318
  - file 280
  - floating point 82
  - integer 82, 280
  - list file 318
  - load all files 318
  - load modules 263
  - machine 82
  - path specification 265
  - replace file 318
  - scanned 263

- single precision 280
  - Standard ANSI 280
  - version 280
- line number
  - information 473
- link
  - command file 264
  - user command file 22
- linker
  - # character prefix,comment 263
  - build freestanding program 260
  - clnk 9
  - command file 262
  - command file example 298
  - command item 262
  - comment 263
  - global command line options 265
  - output file 261
  - physical memory 261
- list directive 226
- listing
  - cross reference 20
  - file location 28
  - file path specification 314
  - interspersed C and assembly file 19
- lit directive 227
- literals
  - in @near space 63
- local
  - labels 58
- local directive 184, 228
- local variable
  - reorder 469
- locate source file 307
- log function 133
- log10 function 134
- long multiplication 402

## M

- macro
  - exit 190
  - expansion in listing 190

---

- internal label 184
- named syntax 189
- numbered syntax 189
- macro directive 229
- main
  - function 295
- main() routine 35
- map
  - file description 295
  - modules section 295
  - produce information 265
  - segment section 295
  - stack usage section 295
  - symbols section 295
- max function 135
- memchr function 136
- memcpy function 137
- memcpy function 138
- memmove function 139
- memory
  - location 46
  - long range 50
  - mapped I/O 46
- memory models 12, 39
- memset function 140
- messg directive 231
- mexit directive 230, 232
- min function 141
- mlist directive 233, 246
- modf function 142
- Motorola
  - S-Records format 311
  - standard S-record, generating 23
- moveable
  - code section 284
  - function used 285
- moveable code segment 111

## N

- named syntax, example 230, 241
- new
  - segment control 262

- start region 273
- nolist directive 234
- nopage directive 235
- numbered syntax, example 230, 241

## O

- object
  - file location 28
  - image 259
  - module 260
  - module inspector 10
  - relocatable 320
  - relocatable file output 181
  - relocatable file size 320
  - size 320
- offset
  - segment parameter 277
  - setting 278
- offset directive 236
- optimization
  - disable selectively 476
  - keep line 476
  - specific code 475
- option
  - global 264
- org directive 237
- output
  - default format 311
  - file name 264
  - listing line number 467
  - specify format 144
- override
  - data bias 310
  - text bias 310

## P

- page
  - address extension 187
  - value 187, 312
- page directive 238
- page header 253
- paginating output 307

---

- parser
  - behaviour 467
  - compiler pass 464
  - error log file 466
- plen directive 239
- pointer
  - narrow 468
- pow function 143
- prefix
  - filename 469
  - modules 318
- preprocessor
  - #define 464
  - #include 464
  - run only 466
- printf function 144
- private name region
  - use 290
- program
  - stop execution of 108
- putchar function 149
- puts function 150

**R**

- rand function 151
- range specification 307
- redirect output 307
- REFs 289
- region
  - name 262
  - private 273
  - public 273
  - use of private name 290
- register
  - input/output 48
- relative address 302
- repeat directive 240
- repeatl directive 241
- restore directive 243
- rexit directive 241, 244
- ROM 46
- runtime startup

- modifying 34

## S

- save directive 245
- section
  - .bit 21, 42
  - .bsct 21, 49
  - .bss 21, 50
  - .const 21
  - .data 21, 50
  - .eeprom 21, 51
  - .fconst 21
  - .fdata 21
  - .info. 180, 181, 265, 320
  - .text 21
  - .ubsct 21, 49
- assembler directive 247
- crossing boundary 41
- curly braces,initialized data 53
- definition 259
- name 54
- parenthesis,code 53
- pragma definition 53
- pragma directive 54
- single 474
- square brackets, uninitialized data 53
- unused 268
- user defined 53

sections

- default 53
- predefined 53
- relocation 277

segment

- .debug 296
- .info. 297
- bsct start address 271
- bss start address 271
- build new 280
- control options 264, 267
- data start address 271
- definition 259
- fill 267

---

- follow current 267
- maximum size 268
- name 269
- overlap checking 270, 277
- overlapping 280
- overlapping control 270
- root 268
- round up address 269
- section overlap 271
- space name 277
- start,new 267
- text start address 271
- zero size 263
- separated address space 277
- set directive 249
- share
  - local variable 468
- short addressing 12
- short range
  - @tiny modifier 49
  - section 198, 322, 323
  - size 39, 40, 49
  - space 323
- sin function 152
- sinh function 153
- source files listing 306
- source listings 306
- space
  - for function 467
  - for variable 467
- space name
  - definition 270
- spc directive 250
- sprintf function 154
- sqrt function 155
- srand function 156
- stack
  - amount of memory 295
  - check overflow 473
  - need 295
- stack model
  - long 40, 50
  - long,64K 39
  - short 39
- stack pointer 35
- standard ANSI libraries 280
- startup file
  - crt.s 34
  - crti.s 38
- static data 304
- STM8
  - addressing mode 183
  - instruction set 183
- ST-Microelectronics syntax 182
- strcat function 157
- strchr function 158
- strcmp function 159
- strcpy function 160
- strncpy function 161
- strings 164
- strlen function 162
- strncat function 163
- strncmp function 164
- strncpy function 165
- strpbrk function 166
- strrchr function 167
- strspn function 168
- strstr function 169
- strtod function 170
- strtol function 171
- strtoul function 172
- suffix
  - assembly file 74
  - C file 74
  - input 315
  - output 315
- suffix letter 185
- suppress pagination 307
- switch directive 251
- symbol
  - \_\_endmem 66
  - \_\_startmem 66
  - \_endmem 67
  - \_startmem 67

---

- alias 291
- define 262
- define alias 274
- define new 274
- definition 274
- export 294
- initialization image start, equal 275
- logical end value, equal 275
- logical start value, equal 275
- physical end value, equal 275
- physical start value, equal 275
- size value, equal 275
- sort alphabetically 265
- sort by address 265
- user-defined 466

- symbol table
  - add 274
  - information 320
  - new 289

## T

- tab character setting 252
- tabs directive 252
- tan function 173
- tangent 173
- tanh function 174
- task entries 295
- title directive 253
- tolower function 175
- toupper function 176
- translate executable images 310

## U

- unreachable code
  - eliminate 11

## V

- variable
  - reorder local 469
- vers option 6
- volatile
  - data 43

- qualifier 43
  - using keyword 43

## W

- warnings 79, 469
- window
  - set shift 265, 324
  - size 270

## X

- xbit
  - assembler directive 193
- xbit.b
  - assembler directive 193
- xdef directive 255, 256
- xref directive 254, 255, 256