Version 4.8

C Cross Compiler User's Guide for NXP HC12/HCS12

Table of Contents

Preface

Chapter 1
Introduction

Chapter 2
Tutorial Introduction

Chapter 3
Programming Environments

Chapter 4
Using The Compiler

Chapter 6
Using The Linker

Chapter 7
Debugging Support

Chapter 8
Programming Support

Chapter A
Compiler Error Messages

Chapter B
Modifying Compiler Operation

Chapter C
HC12/HCS12 Machine Library

Chapter D
Compiler Passes

Preface

The *Cross Compiler User's Guide for HC12/HCS12* is a reference guide for programmers writing C programs for HC12/HCS12 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 HC12/HCS12 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 HC12/HCS12 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, "*HC12/HCS12 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.

CHAPTER

1

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

The C cross compiler targeting the HC12/HCS12 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 **cx6812** 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 appears as:

```
cx6812 +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:

```
cx6812 +debug file.c
```

In this example, `cx6812 +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|h12]
```

means that one file with the extension `.o` or `.h12` 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:

```
program_name [<flags>] <files>
```

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
-a	accept a long integer value
-b	accept a long integer value
-e	accept a long integer value
-f	accept a single character
-h	simply a flag indicator
+h	accept a character string
-m	accept a short integer value
-n	accept a character string and may be repeated
-o	accept a character string
-p	simply a flag indicator
-pl	accept a long integer value
-pn	simply a flag indicator
-pp	simply a flag indicator
-s	simply a flag indicator
-w	simply a flag indicator
-x	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. **cx6812** controls the operation of these programs automatically, using the options you specify, and runs the programs described below in the order listed:

cp6812 - the C preprocessor and language parser. *cp6812* expands directives in your C source and parses the resulting text.

cg6812 - the code generator. *cg6812* accepts the output of *cp6812* and generates assembly language statements.

co6812 - the assembly language optimizer. *co6812* optimizes the assembly language code that *cg6812* generates.

ca6812 - the assembler. *ca6812* converts the assembly language output of *co6812* 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:

bit 0	set if nowiden option specified (**+nowiden**)
bit 1	set if single precision option specified (**+sprec**)
bit 2	set if unsigned char option specified (**-pu**)
bit 3	set if alignment option specified (**+even**)
bit 4	set if reverse bitfield option specified (**+rev**)
bit 5	set if no enum optimization specified (**-pne**)
bit 6	set if no bitfield packing specified (**-pnb**)
bit 7	set if no constant replacement specified (**-pnc**)
bit 8	set if no constant propagation specified (**-pcp**)
bit 9	set if prototype checking specified (**+proto**)

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 **cx6812**, 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 listed below to inspect the executable.

cbank - optimize the bank filling with object file. It reorganizes a object list in order to fill as completely as possible the smallest amount of banks and produces as result a text file containing the object file names in the proper order.

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 to 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.

cv695 - IEEE695 format converter. *cv695* allows you to generate IEEE695 format file. This utility must to be used only after the linker.

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 uses registers **d and x** to hold the first argument of a function call if:

  1) the function does not return a structure or a double, and

  2) the first argument is derived from one of the following types:

  char,
  short,
  int, long,
  float,
  pointer to...,
  or *array of....*

- 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 Bank Switching

The compiler supports bank switching for **code** and **data**, using the internal window mechanism provided by the **HC12/HCS12** processor. Bank switching is supported via:

- **@far** type qualifier to describe a function relocated in a different bank. Calling such a function implies a special calling sequence, and a special return sequence. Such a function has to be defined **@far** and referenced as **@far** in all the files using it. The compiler also provides a specific option **+modf** to automatically consider all the functions to be **@far**. The **@far** type modifier is also used to declared variables allocated in a data bank.

- Linker options are required to ensure proper physical and logical addresses computations. The linker is also able to automatically fill banks without any need to take care of the page boundaries.

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 HC12/HCS12 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* ***0x***. *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.

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*.

CHAPTER

2

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:

Acia.c, Example file

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

```
/*    EXAMPLE PROGRAM WITH INTERRUPT HANDLING
 *    Copyright (c) 1998 by COSMIC Software
 */
#include <ioa4.h>

#define SIZE512          /* buffer size */
#define TDRE0x80         /* transmit ready bit */

/*    Authorize interrupts.
 */
#define cli()_asm("andcc #$EF\n")

/*    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 0.
 */
void outch(char c)
      {
      while (!(SC0SR1 & TDRE))/* wait for READY */
            ;
      SC0DRL = c;/* send it */
      }
```



```
/*    Character reception routine.
 *    This routine is called on interrupt.
 *    It puts the received char in the buffer.
 */
@interrupt void recept(void)
      {
      SC0SR1;                 /* clear interrupt */
      *ptecr++ = SC0DRL;      /* 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 */
      SC0BDL = 52;            /* initialize SCI */
      SC0CR2 = 0x2c;          /* parameters for interrupt */
      cli();                  /* 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.

As it processes the command line, **cx6812** 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:

file.s	Assembler source module
file.o	Relocatable object module
file.h12	input (*e.g.* libraries) or output (*e.g.* absolute executable) file for the linker

Compiling and Linking

To compile and assemble *acia.c* using default options, type:

```
cx6812 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**.

```
cx6812 -s acia.c
```

The **-s** option directs **cx6812** to stop after having produced the assembly file *acia.s*. You can then edit this file with your favorite editor. You can also visualize it with the appropriate system command (*type*, *cat*, *more*,...). For example under MS/DOS you would type:

```
type acia.c
```

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

```
cx6812 -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:

```
cx6812 -v -s acia.c
```

the display will look like something similar to the following:

```
acia.c:
        cp6812 -o \2.cx1 -i\cx\h6812 -u acia.c
        cg6812 -o \2.cx2 \2.cx1
        co6812 -o acia.s \2.cx2
```

The compiler runs each pass:

cp6812	the C parser
cg6812	the assembly code generator
co6812	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*.

```
cx6812 acia.s
```

or

```
ca6812 -i\cosmic\h6812\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:

```
ca6812 -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.h12** 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
.ftext	executable code in paged area (**@far**)
.text	code (or program) section (**@near**)
.const	constant and literal data (*e.g.* **ROM**)
.fdata	large initialized variables (**@far**)
.data	initialized static data
.bss	all non initialized static data
.bsct	initialized data in the *direct page* (see **@dir** in chapter 3)
.ubsct	non initialized data in the *direct page*
.fbss	large non initialized variables (**@far**)
.eeprom	any variable in eeprom (**@eeprom**)

When the **+ceven** option is selected, the *constant* section is splitted in two parts:

Type	Description
.const	single byte constants
.const.w	word aligned constants

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) 1995 by COSMIC Software
line 3 #
line 4 +seg .const -b0x1c000 -o0xc000 -n.const # const
unbanked
line 5 +seg .const -a.text          # page 7 unbanked
line 6 +seg .data -b 0x2000         # data start address
line 7 +def __sbss=@.bss            # start address of bss
line 8 crts.o                       # startup routine
line 9 acia.o                       # application program
line 10 \cx\lib\libi.h12            # C library (if needed)
line 11 \cx\lib\libm.h12            # machine library
line 12 +seg .vector -b0x1ffb8 -o0xffb8# vectors start
line 13 vector.o                    # interrupt vectors file
line 14 +def __memory=@.bss         # symbol used by startup
line 15 +def __stack=0x1000         #stack ptr initial value
```

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 .const -b0x1c000 -o0xc000 -n.const` creates a const segment located at physical address `1c000` (hexa) and logical address `c000` which is named *.const* (page 7 unbanked).

line 5: `+seg .text -a.const` creates a text (code) segment located after the previous const segment (page 7 unbanked).

line 6: `+seg .data -b0x2000` creates a data segment located at `2000` (hex address)

line 7: `+def __sbss=@.bss` defines a symbol *__sbss* equal to the value of the current address in the **.bss** segment. This is used to get the address of the start of the *bss*. The symbol *__sbss* is used by the startup routine to reset the *bss*.

line 8: `crts.o` runtime startup code. It will be located at `0xc000` (code segment)

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

line 10: `libi.h12` the integer library to resolve references

line 11: `libm.h12` the machine library to resolve references

line 12: `+seg .vector -b0x1ffb8 -o0xffb8` creates a new constant segment located at `ffb8`

line 13: `vectors.o` interrupt vectors file

line 14: `+def __memory=@.bss` defines a symbol *__memory* equal to the value of the current address in the **.bss** segment. This is used to get the address of the end of the *bss*. The symbol *__memory* is used by the startup routine to reset the *bss*.

line 15: `+def __stack=0x4000` defines a symbol *__stack* equal to the absolute value `4000` (hex value). The symbol *__stack* is used by the startup routine to initialize the stack pointer.

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

The *crts.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.h12 acia.lkf
```

Step 4: Generating S-Records file

Although *acia.h12* 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.h12* to *Motorola standard S-record* format:

```
chex acia.h12 >acia.hex
```

or

```
chex -o acia.hex acia.h12
```

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 about the converter, see **Chapter 8**, "*The chex Utility*".

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 .bsct -b0x0                # zpage start address
var_zpage.o                     # file with zpage variable
+seg .const -b0x1c000 -o0xc000 -n.const# const unbanked
+seg .text -a .const -n.common# page 7 unbanked
+seg .data -b 0x2000            # data start address
+seg .bss -b 0x2500             # bss start address
+def __sbss=@.bss               # symbol used by startup
crts.o                          # startup routine
acia.o                          # main program
module1.o                       # application program
+seg .text -b 0x00000 -o0x8000# start new text section
module2.o                       # application program
module3.o                       # application program
+seg .text -a .common -n .common7 -it# Page 7 unbanked
\cx\lib\libi.h12                # C library (if needed)
\cx\lib\libm.h12                # machine library
+seg .vector -b0x1ffb8 -o0xffb8# vectors start
vector.o                        # interrupt vectors
+def __memory=@.bss             # symbol used by startup
+def __stack=0x4000             # stack pointer initial value
```

In this example the linker will locate and merge *crts.o*, *acia.o* and m*odule1.o* in a *text* segment at **0xc000** (page 7, unbanked), a *data* segment at **0x2000** and a *bss* segment, if needed at **0x2500**. *zero page* variables will be located at **0x0.** The rest of the application, *module2.o* and *module3.o* will be located in a bank at **0x8000**. The libraries will be located and merged in the page 7 (unbanked) segment named *.common* at **0xc000** then the interrupt vectors file, *vector.o* in a *.vector* segment at **0xffb8**.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 **.data** section and

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

```
char var2;
```

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

There is one exception to the above rules when you declare data that will be located in the **zero page**, using the **@dir** type qualifier. In this case, the data is generated into the **.bsct** section if it is initialized or generated into the **.ubsct** 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 **crtsi.h12** 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.

```
+seg .text -b 0xfe000 -o 0xe000 -n.text# program start
+seg .const -a .text           # constant follow code
+seg .bsct -b 0 -m 0x100       # zpage start address
+seg .data -b0x2000            # data start address
+def __sbss=@.bss              # symbol used by startup
\cx\lib\crtsi.h12              # startup with auto-init
acia.o                         # main program
module1.o                      # module program
\cx\lib\libi.h12               # C library (if needed)
\cx\lib\libm.h12               # machine library
+def __memory=@.bss            # symbol used by library
+def __stack=0x4000            # stack pointer initial value
```

In the above example, the *text* segment is located at address **`0xe000`**, the *data* segment is located at address **`0x2000`**, immediately followed by the *bss* segment that contains uninitialized data. The copy of 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:

```
+seg .text -b 0xfe000 -o0xe000 -n.text# program start
+seg .const -a .text           # constant follow code
+seg .bsct -b 0 -m 0x100       # zpage start address
+seg .data -b0x2000            # data start address
+def __sbss=@.bss              # symbol used by startup
\cx\lib\crtsi.h12              # startup with auto-init
acia.o                         # main program
module1.o                      # module program
+seg .text -b0xff000 -o0xf000 # new code segment
module2.o                      # module program
module3.o                      # module program
\cx\lib\libi.h12               # C library (if needed)
\cx\lib\libm.h12               # machine library
+seg .vector -b 0x1ffb8 -o0xffb8# vectors start
vector.o                       # interrupt vectors
+def __memory=@.bss            # symbol used by startup
+def __stack=0x4000            # stack pointer initial value
```

or


```
+seg .text -b 0xfe000 -o0xe000 -n .text# program start
+seg .const -a .text            # constant follow code
+seg .bsct -b 0 -m 0x100        # zpage start address
+seg .data -b0x1000             # data start address
+def __sbss=@.bss               # symbol used by startup
\cx\lib\crtsi.h12               # startup with auto-init
acia.o                          # main program
module1.o                       # module program
+seg .text -b0xff000 -o0xf000 -it# set the section attribute
module2.o                       # module program
module3.o                       # module program
\cx\lib\libi.h12                # C library (if needed)
\cx\lib\libm.h12                # machine library
+seg .vector -b 0x1ffb8 -o0xffb8# vectors start
vector.o                        # interrupt vectors
+def __memory=@.bss             # symbol used by startup
+def __stack=0x4000             # stack pointer initial value
```

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 `0xf000`.

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 **cx6812** to control the compilation process.

To compile and produce a relocatable file named *acia.o*, type:

```
cx6812 acia.c
```

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:

```
cx6812 -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 respectivally the **-cl** and **-co** options:

```
cx6812 -cl\mylist -co\myobj 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.

cx6812 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:

```
cx6812 -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.

CHAPTER

3

Programming Environments

This chapter explains how to use the COSMIC program development system to perform special tasks required by various HC12/HCS12 applications.

Introduction

The HC12/HCS12 COSMIC compiler is an ANSI C compiler that offers several extensions which support special requirements of embedded systems programmers. This chapter provides details about:

- Modifying the Runtime Startup
- Initializing data in RAM
- The const and volatile Type Qualifiers
- Performing Input/Output in C
- Placing Data Objects in The Bss Section
- Placing Data Objects in The Zero Page Section
- Placing Data Objects in the EEPROM Space
- Referencing Absolute Addresses
- Redefining Sections
- Inlining Functions
- Optimizing boolean functions
- Accessing Internal Registers
- Inserting Inline Assembly Instructions
- Referencing Absolute Addresses
- Writing Interrupt Handlers
- Placing Addresses in Interrupt Vectors
- Calling a Bank Switched Function
- Accessing Banked Data
- Using Position Independent Code

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,
- *_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 **crts.h12** included on your distribution media. It does not perform automatic data initialization. A special startup program is provided, **crtsi.h12**, which is used instead of *crts.h12* when you need 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.

Description of Runtime Startup Code

```
;    C STARTUP FOR MC68HC12
;    Copyright (c) 1996 by COSMIC Software
;
     xdef  _exit, __stext
     xref  _main, __sbss, __memory, __stack
;
__stext:
     clra              ; reset the bss
     clrb
     ldx   #__sbss     ; start of bss
     bra   loop        ; start loop
zbcl:
     std   2,x+        ; clear word
loop:
     cpx   #__memory; up to the end
     blo   zbcl        ; and loop
```



```
17     lds    #__stack    ; initialize stack pointer
18 ifdef PIC
19     lbsr   _main
20 else
21     jsr    _main       ; execute main
22 endif
23 _exit:
24     bra    _exit       ; stay here
25 ;
26     end
```

_main is the entry point into the user C program.

__memory is an external symbol defined by the linker as the end of the **bss** section. The start of the *bss* section is marked by the external symbol **__sbss**.

__stack is an external symbol defined by the linker as an absolute value.

Lines 8 to 16 reset the *bss* section. Note that the usage of a 16 bit instruction **std** implies that either the *bss* segment is not followed by any significant data, or its size is even. The default placement locates the *bss* segment at the end of the data area, but if such an allocation is modified, the *bss* segment must be linked with option **-r1** to ensure an even size if another segment is allocated just after.

Line 17 sets the stack pointer. You may have to modify it to meet the needs of your application.

Line 21 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 subdirectory.

```
;       C STARTUP FOR MC68HC12
;       WITH AUTOMATIC DATA INITIALISATION
;       Copyright (c) 2000 by COSMIC Software
;
        xdef    _exit, __stext
        xref    _main, __sbss, __memory, __idesc__, __stack
;
__stext:
        lds     #__stack        ; initialize stack pointer
ifdef PIC
        leax    __idesc__,pcr; descriptor address
ifdef MCX
        subx    #__idesc__      ; code offset
        pshx                    ; on the stack
else
        tfr     x,d             ; compute
        subd    #__idesc__      ; code offset
        pshd                    ; on the stack
endif
else
        ldx     #__idesc__      ; descriptor address
endif
        ldy     2,x+            ; start address of prom data
ibcl:
        ldaa    5,x+            ; test flag byte
        beq     zbss            ; no more segment
        bpl     nopg            ; page indicator
        leax    2,x             ; skip it
nopg:
        bita    #$60            ; test for moveable code segment
        bne     dseg            ; no, copy it
        ldy     -2,x            ; reload code address
        bra     ibcl            ; and continue with next segment
```



```
dseg:
       pshx                 ; save pointer
       tfr    y,d           ; start address
       subd   -2,x          ; minus end address
       ldx    -4,x          ; destination address
ifdef PIC
ifdef MCX
       addy   2,s           ; adjust code offset
else
       exg    d,y           ; adjust
       addd   2,s           ; code address
       exg    d,y
endif
endif
dbcl:
       movb   1,y+,1,x+     ; copy from prom to ram
       ibne   d,dbcl        ; count up and loop
ifdef PIC
ifdef MCX
       suby   2,s           ; restore code address
else
       exg    d,y           ; restore
       subd   2,s           ; code address
       exg    d,y
endif
endif
       pulx                 ; reload pointer to desc
       bra    ibcl          ; and loop
zbss:
       ldx    #__sbss       ; start of bss
       clrb                 ; complete zero
       bra    loop          ; start loop
zbcl:
       std    2,x+          ; clear byte
loop:
       cpx    #__memory     ; end of bss
       blo    zbcl          ; no, continue
ifdef PIC
       puld                 ; clean stack
       lbsr   _main         ; execute main
else
       jsr    _main         ; execute main
endif
_exit:
       bra    _exit         ; stay here
;
       end
```

crtsi.s performs the same function as described with the *crts.s*, but with one additional step. Lines (marked in bold) in *crtsi.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.

For more information, see “*Generating Automatic Data Initialization*” in **Chapter 2** and “*Automatic Data Initialization*” in **Chapter 6**.

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 PORTD @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 data object by the function call. 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 indeterminate.

You may specify *const* and *volatile* together, in either order. A *const volatile* data object could be a Read-only status register, or a 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);
```

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, **crts.s** and **crtsi.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 beeing explicitely initialized to zero.

Placing Data Objects in The Zero Page Section

The **zero page** section, or "**zpage**", refers to data that is accessed in the internal memory of the HC12/HCS12 chip and may be accessed with one byte address; this is the first 256 bytes of memory. Placing initialized data objects in the *zero page* section optimizes code size and execution time.

To place data objects selectively into the *zero page* section, use the type qualifier **@dir** when you declare the data object. For example:

```
@dir char var;
```

A data object declared this way will be located into the section **.bsct**, if it is initialized, or in the section **.ubsct** otherwise. An external object name is published via a **xref.b** declaration at the assembly language level.

To place data objects into the *zero page* on a file basis, you use the **#pragma** directive of the compiler. The compiler directive:

```
#pragma space [] @dir
```

instructs the compiler to place all data objects of storage class **extern** or **static** into the *zero page* for the current unit of compilation (usually a file). The section must end with a **#pragma space []**.

The compiler provides a special option, **+zpage**, which forces the **#pragma** directive described above for all files compiled with that option.

NOTE

The code generator does not check for zero page overflow.

Placing Data Objects in the EEPROM Space

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

```
@eeprom char var;
```

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

```
#pragma space [] @eeprom
```

instructs the compiler to treat all *extern* and *static* data in the current file as *eeprom* locations. The section must end with a **#pragma space []**.

NOTE

*The library modules handling the specific eeprom control registers, the application must include in, at least one C source file one of the provided header files specific to the actual target (for instance, iosdp256.h). Otherwise, the missing symbols will have to be manually defined with **+def** directives in the linker command file.*

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

```
+seg .eeprom -b0x1000 -m4096
var_eeprom.o
```

will create a segment located at address **0x1000**, with a maximum size of 4096 bytes.

NOTE

The code generator cannot check if the final address of an @eeprom object will be valid after linkage.

You must then link with the proper libraries. For more information, see "*Linking Library Objects*" in **Chapter 6**.

Redefining Sections

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

Section	Description
.ftext	executable code in paged area (@**far**)
.text	executable code in common area
.const	text string and constants
.fdata	variable in paged area (@**far**)
.data	initialized variables
.bss	uninitialized variables
.bsct	variable in zero page (@**dir**)
.ubsct	uninitialized variables in zero page (@**dir**)
.eeprom	variable in eeprom (@**eeprom**)
.feeprom	@far variable in eeprom (@**far** @**eeprom**)

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
@dir
@eeprom
@near
@far

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.

When the **+ceven** option is selected in order to have two different sections for aligned and non aligned constants, renaming the **const** section renames both sections by applying the suffix **.w** to the word aligned part.

The following directives:

```
#pragma section (code)
#pragma section const {string}
#pragma section [udata]
#pragma section {idata}
#pragma section @dir {zpage}
#pragma section @eeprom {e2prom}
#pragma section @far {dpage}
```

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**
- zerodirect page variables are redirected to section **.zpage**
- eeprom variables are redirected to section **.e2prom**
- paged variables are redirected to section **.dpage**

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

The following directive:

```
#pragma section ()
```

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

Inlining Functions

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 HC12/HCS12 processor, using **@inline** functions. Such functions shall be declared as external functions with the *@inline* modifier. The compiler recognizes two predefined functions when declared as follows:

```
@inline char carry(void);
@inline char overflow(void);
```

carry the *carry* function is used to test or get the carry bit from the condition register. If the *carry* function is used in a test, the compiler produces a **bcc** or **bcs** instruction. If the *carry* function is used in any other expression, the compiler produces a code sequence setting the **b** register to 0 or 1 depending on the carry bit value.

overflow the *overflow* function is used to test or get the overflow bit from the condition register. If the *overflow* function is used in a test, the compiler produces a **bvc** or **bvs** instruction. If the *overflow* function is used in any other expression, the compiler produces a code sequence setting the **b** register to 0 or 1 depending on the overflow bit value.

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

Optimizing boolean functions

When a function returns a value used as a boolean, the compiler tests the content of the **d** register to setup the flags and perform the conditional branch. If the function is declared with the **@bool** type modifier, the compiler assumes that the flags are correctly set by the called function. It does not test the register and directly performs the conditional branch. This feature is useful for several library functions which return boolean values, and which are coded in assembler, thus already setting the flags correctly. This extension can be used on a C function. In this case, the compiler modifies the return sequence to set the flags appropriately before returning to the caller.

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 **0x0** with the identifier name *PORTA*, write a definition of the form:

```
char PORTA @0x0;
```

where **@0x0** indicates an absolute address specification and not a data initializer. Since input/output on the HC12/HCS12 architecture is memory mapped, performing I/O in this way is equivalent to writing in any given location in memory.

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

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

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

```
#define PORTA *(char *)0x0
```

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.

It is also possible to define structures at absolute addresses. For example, one can write:


```
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 HC12/HCS12 memory map.

Accessing Internal Registers

All the I/O registers are declared in the **io.h** files provided with the compiler, depending on the specific derivative. Such a file should be included by a:

```
#include <iosdp256.h>       /* for MCS912DP256 */
```

in each file using the input-output registers. All the register names are defined by assembly *equates* which are made *public*. This allows any assembler source to use directly the input-output register names by defining them with an *xref* directive. All those definitions are already provided in the **io.s** files which may be included in an assembly source by a:

```
    include "iosdp256.s"     ; for MCS912DP256
```

All these header files assume a default location for the input-output registers depending on the actual target. This default value may be changed by defining the C symbol **_BASE** by a **#define** directive before the header file # include:

```
#define _BASE 0x1000
#include <iosdp256.h>
```

The default value of **0** for the register starting address as defined by the file <*iosdp256.h*> is changed to **0x1000**.

Header files specific to **HCS12** (**S12**) family members start with “**ios**” while files specific to the standard **HC12** family members start with “**io**”.

If these compiler provided header files are not used, the compiler may still need to access some registers (**PPAGE**, **DPAGE**, **EPAGE**), and will use the code generator options **-t** and **-r** to locate them properly.

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
#pragma asm	start assembler block
#pragma endasm	end assembler block

The compiler also accepts shorter sequences with the same meaning:

Directive	Description
#asm	start assembler block
#endasm	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 { */
     sec         ; set carry bit
     rol asmvar ; access assembler variable
#endasm
     else
          test = 1;
     }
```

NOTE

Preprocessing directives are still handled inside assembly code, but #define symbols or macros are not replaced within assembly instruction and operands by default. In order to enable such a replacement in the assembly code, the compiler must be run with the ***-pad*** *option.This expansion is limited to the simple macros (without arguments).*

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.

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().*

The string you specify follows standard C rules. For example, carriage returns can be denoted by the '\n' character.

To produce the following assembly sequence:

```
leas $1000,x
jsr _main
```

you would write in your C program:

```
_asm("leas $1000\njsr _main");
```

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

To copy a value in the condition register, you write:

```
_asm("tfr b,ccr", varcc);
```

The *varcc* variable is passed in the **d** register, as a first argument. The *_asm* sequence then transfers the low byte from the **b** register to the condition register.

_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:

```
if (_asm("tfr ccr,b\n") & 0x02)
```

allows to test the overflow bit. That way, you can use *_asm()* to write equivalents of C functions directly in assembly language.

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 compiler is run with the **-ga** option.*

By default, *_asm()* is returning an **int** as any undeclared function. To avoid the need of several definitions (usually confictuous) 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
     ldab #7
$N:
     dbne b,$L ; loop on the previous label
#endasm
```

This syntax can also be used as a prefix if labels are interspersed. This allows the function to be called and inlined multiple times without creating duplicate labels or conflicts.

```
#asm
     addd #0
     bne  $N_1
     clrb
     bra  $L_3
$L_1:
     lsrd
     bcc  $L_2
     ldd  #0xF000
$L_2:
     idiv
$L_3:
#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 any 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 "**rti**" instruction for the return sequence.

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)
        {
        ...
        }
```

You can call an *@interrupt* function directly from a C function. The compiler will simulate an interrupt stack frame before jumping in the interrupt function in order to allow the **rti** instruction to return properly.

NOTE

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

Due to the HC12/HCS12 interrupt mechanism, an interrupt function cannot be directly placed in a bank. It should be normally located in the common part and then explicitly defined with the **@near** modifier if the source file is compiled with the **+modf** option.

Placing Addresses in Interrupt Vectors

You may use either an assembly language program or a C program to place the addresses of interrupt handlers in interrupt vectors. The assembly language program would be similar to the following example:

```
        switch .const
        xref handler1, handler2, handler3
vector1:dc.w handler1
vector2:dc.w handler2
vector3:dc.w handler3
        end
```

where *handler1* and so forth are interrupt handlers.

A small C routine that performs the same operation is:

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

where *handler1* and so forth are interrupt handlers. Then, at link time, include the following options on the link line:

```
+seg .vector -b0xfffce -o0xffce vector.o
```

where *vector.o* is the file which contains the vector table. This file is provided in the compiler package. You should modify this vector table as necessary for your application.

Calling a Bank Switched Function

When using the **HC12/HCS12** bank switching mechanism, it is possible to call directly a function which is located in a different bank. To perform the correct call, it is necessary to declare the function with the **@far** type modifier.

NOTE

The libraries are **not** built as *@far* functions and should **not** be located in a banked area, if they need to be accessed from any bank.

It is possible to force the compiler to build all the functions as *@far* functions by using the **+modf** option. An example of such a definition is:

```
@far int func(void)
     {
     ...
     }
```

When linking a bank switched application, several options must be used to configure the linker properly:

-b	should be specified with the **physical** address for each code segment or for the first bank if the **-w** option is used.
-bs	is automatically set with the value **14** for the **HC12/HCS12** processor. The bank number extracted by the linker and copied into the window base register, then points to a **16K** bytes block. This option is located on the command line.
-m	should be specified with the maximum size of each segment, or the maximum size of all the banks if the **-w** option is used.
-o	should be specified with the **logical** starting address for each code segment or bank. It normally is the window base address in the **64K** limits. It should be **0x8000** for the **HC12/HCS12**.
-w	should be specified with the window size to allow the linker to build automatically banked segments. It should be **0x4000** for the **HC12/HCS12**.

Assuming we are building an application with a root segment at `0xC000` and a window at `0x8000`, the link command file should look like:

```
+seg .text -b 0x10000 -o 0x8000 -m 0x4000
func1.o func2.o func3.o
+seg .text -b 0x14000 -o 0x8000 -m 0x4000
func4.o func5.o func6.o
+seg .text -b 0x1c000 -o 0xc000
main.o libm.h12
```

```
+seg .text -b 0x7f0000 -o 0x8000 -m 0x4000
func1.o func2.o func3.o
+seg .text -b 0x7f4000 -o 0x8000 -m 0x4000
func4.o func5.o func6.o
+seg .text -b 0x7fc000 -o 0xc000
main.o libm.h12
```

given two banks, the first one obtained from *func1*, *func2* and *func3* linked at physical address `0x10000`, the second obtained from *func4*, *func5* and *func6* linked at physical address `0x14000`. The window mechanism has to be initialized with the first window at `0x8000`. The code to perform this initialization has to be located in the root segment, for instance at the beginning of the main function. The linker should thus be called with the following options:

```
clnk -o appli.h12 -bs14 appli.lkf
```

NOTE

*Applications **not** using bank switching **should** specified the **-bs0** option to disabled the internal banking verification.*

It is possible to let the linker automatically fill consecutive banks by using the **-w** option specifying the window size. In that case, the **+seg** directive describes the first bank and if a new object file turns the bank size large than the window size, a new bank is automatically starting from a physical address obtained by adding the window size to the physical starting address of the previous bank. The **-m** option specifies the maximum size of all the possible banks. With the following link command file:

```
+seg .text -b 0x10000 -o 0x8000 -m 0x8000 -w 0x4000
func1.o func2.o func3.o
func4.o func5.o func6.o
+seg .text -b 0xc000 -o 0xc000
main.o libm.h12
```

a new segment will be started automatically at physical address **`0x14000`** from the first object module which turns the bank size larger than 16K. The new bank restarts with the same logical address than the previous one, and the maximum size is adjusted by substracting the window size to the value found in the previous bank.

Because code and data spaces are using different chip selects, the resulting physical addresses may overlap while they do not in fact address the same memory space. To allow the linker to verify properly any possible overlapping, segments belonging to the same memory kind can be grouped together with a space name defined on the segment opening directive by using the **-s** option followed by an arbitrary space name. The linker will verify overlapping between segments sharing the same space name.

The linker also verifies that a bank is properly entered with a **call** instruction. Any attempt to enter a bank with a **jsr** instruction will be reported as an error, unless the **jsr** is issued from the same bank.

For more information, see “*Bank Switching*” in **Chapter 6**.

Accessing Banked Data

The **68HC12A4** is able to extend the data range by using a bank mechanism similar to the code banking. Two areas are available for data banking, the first one located from `0x7000` to `0x7fff` (4K bytes) using the **DPAGE** register, the second one located from `0x0000` to `0x03ff` or `0x0400` to `0x07ff` (1K bytes) using the **EPAGE** register.

A variable can be defined in a data bank by using the **@far** modifier on its declaration. By default, such a variable is located in the **DPAGE** area. To access the **EPAGE** area, the **@epage** modifier has to be used along with *@far*. For examples:

@far int i;	located in DPAGE area
@epage @far int j;	located in EPAGE area

A constant can also be defined in a code bank by using the @far modifier along with the @ppage modifier. Such a constant is accessed using the **PPAGE** register. This register beeing used implicitly by any banked code, such a constant can be accessed only by a function located in a common (non banked) area, but declared as an @far function to ensure that the PPAGE register is saved on function entry, and restored on function exit. These constraints are properly checked by the compiler and the linker.

@far @ppage int k;	located in PPAGE area

NOTE

Previous versions of the compiler were using the const keyword instead of the @ppage modifier. The new mechanism allows the const keyword to keep its standard meaning and not interfere with the far pointer behaviour. The previous mechanism can be restored by compiling with the -gcf option.

The **S12** family processors are also able to decode external data using the code banking mechanism, and then using the **PPAGE** register to access such variables. This feature is implemented with the same constraints than @far const objects.

By default, any @far data is produced in a separate section called **.fdata**, regardless of any initialization, and any @far constant is produced in the default **.text** section. When using the **+nofds** option from the command line, @far data are produced in the **.data** or **.bss** sections as plain data, thus requesting @far data to be declared in a separate source file in order to allow a correct linking.

If data banking is used, interrupt functions will have to save and restore these registers if they are used by the interrupt code. The compiler will detect automatically any explicit usage done by the interrupt function itself. If the interrupt function does not use directly those registers but calls any other function, the compiler **will not** save the page registers, to keep efficiency on applications not using data bank switching. If data bank switching is used by the called functions, the **@svpage** modifier has to be used on the interrupt function declaration, such as:

```
@svpage @interrupt void func_it(void)
```

NOTE

No data can be allocated across a page boundary and then far pointers calculations do not update the page number. This means that this feature cannot be used to allocate large arrays whose size is larger than the page size.

Linking banked data sections uses the same directives as code bank switching. Because code and data pages sizes are not identical, an extra option is needed to specify the page size when defining a segment. **DPAGE** segments will use a **-ds12** option while **EPAGE** segments will use a **-ds10** option. To link a **DPAGE** banked segment, you can use the automatic filling option such as:

```
+seg .data -ds12 -b 0x0 -o 0x7000 -m 0x4000 -w 0x1000
data1.o data2.o data3.o
```

The physical address will match the RAM chip address decoded by the **CSD** chip select. The maximum size specified here allows up to 4 pages.

Using Position Independent Code

The compiler has the ability to produce **P**osition **I**ndependent **C**ode using the **pc** relative addressing modes both for function calls and constant data access. The resulting code can then be executed at any location in the 64K address range. This feature is accessed by specifying the **+pic** option on the compiler command line.

NOTE

Bank switching cannot be used with ***PIC*** *code.*

By default, the data sections are still using the standard addressing modes and then are linked to a fixed address. Options **+picd** and **+picds** allow data to be also accessed with **pc** relative addressing mode. With the **+picd** option, any data is mapped to the code section, thus letting the assembler producing an efficient offset encoding. With the **+picds** option, data variables are still allocated in their usual sections, giving a cleaner allocation schema, but a less efficient code.

The startup modules *crts.s* and *crtsi.s* and the libraries are prepared to support **PIC** code, but are packaged in their standard shape. To use the PIC feature, you need to recompile the startup routines and the libraries with the **+pic** option set.

Fuzzy Logic Support

The compiler provides a set of functions packaged in a separate library named **fuzzy.h12**, in order to give a direct access to the specific fuzzy instructions provided by the HC12/HCS12. Those functions are basically provided to be used by third party fuzzy logic software tools but may be used directly by some applications. The compiler does not provide any specific tool to design the data structures needed for using those functions. Refer to the “*HC12/HCS12 Reference Manual*” for more information about the fuzzy support.

The functions provided in the fuzzy library are:

memhc12	fuzzify input variables by using the *mem* instruction
revhc12	evaluate rules by using the *rev* instruction
revwhc12	evaluate rules by using the *revw* instruction
wavhc12	defuzzify outputs by using the *wav* and *ediv* instruction

Note that for keeping efficiency, most of these functions are directly inlined in the code output instead of calling actual functions.

Those functions are more completely documented in the **Chapter 4**, “*Using The Compiler*”.

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 banked function code in the **.ftext** section unless the compiler option **+nofts** has been specified, and the non-banked (common) 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.

Literal data such as strings, float or long constants, and switch tables, are normally generated into the **.const** section. An option on the code generator allows word aligned constants to be produced into the **.const.w** section. Another option on the code generator allows constants to be produced directly in the **.text** section.

The compiler generates initialized data into the **.data** section. 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 normally generated into the **.const** section. Initialized data declared with the **@dir** space modifier will be generated into the **.bsct** section. Uninitialized data are normally generated into the **.bss** section or the **.ubsct** section for **@dir** variables, unless forced to the **.data** section by the compiler option **+nobss**. Far data are generated into the **.fdata** section unless the compiler option **+nofds** has been specified. In such a case, far data are allocated like plain data.

Section	Declaration	Reference
.bsct	@dir char i =2;	xdef
.ubsct	@dir char i;	xdef
.fdata	@far int paged	xdef
.data	int init = 1	xdef
.bss	int uninit	xdef
.ftext	@far char putchar(c);	xdef
.text	char putchar(c);	xdef
.const	const int cinit = 1;	xdef
Any of above	extern int out;	xref

Function calls are performed according to the following:

1) Arguments are moved onto the stack from right to left. Unless the function returns a double or a structure, the first argument is stored in the **d** register if its size is less than or equal to the size of an *int*, or in **d,x** register pair if its type is long or unwidened float.

NOTE

By default, character data is sign extended to short, and floats are extended to doubles. This widening can be disabled at the user's option. In that case, character and float will be left unmodified. If widening is disabled, and the first argument to a function is of type char, and it is stored in a register, then it will be stored in register **b**. *Data of type short, integer and long integer are left unmodified.*

2) A data space address is moved onto the stack if a structure or double return area is required.

3) The function is called via a *jsr _func* instruction, or a *call f_func* if the function is an *@far* function.

4) The arguments to the function are popped off the stack.

Register Usage

Except for the return value, the registers **d**, **x**, **y** and the condition codes are undefined on return from a function call. The return value is in **d** if it is of type char widened to short, short, integer or pointer to.... The return value is in the register **d** and **x** if it is of type long or float. The **d** register holds the low order word.

Stack Model

Because the stack pointer can be used to address directly the stack, no register is dedicated as frame pointer. If automatics are needed, the sequence:

```
leas -<#>,s
```

will reserve <#> bytes onto the stack.

This sequence becomes:

```
pshd
leas -<#>,s
```

if the first argument is in the **d** register as described above.

The stack pointer is set to the beginning of the area reserved for automatic data. This is done because of addressing mode characteristics of the HC12/HCS12. The assembler symbol **OFST** is set to the size of the space needed for automatics; arguments are at **OFST+4,s**, **OFST+6,s**, and so forth. Auto storage is on the stack at **OFST-1,s** and down. If no automatics and no arguments are used, the stack frame is **not** built. To return, the sequence:

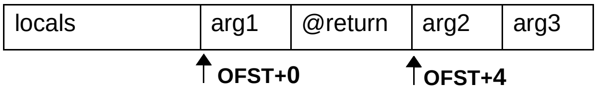
```
leas <#>,s
rts
```

will restore the previous context. Functions that do not have any arguments or autos, and do not use any temporary storage (required to perform operations on structure data or cast float data, for example) do not reference the frame pointer **x** and do not stack it.

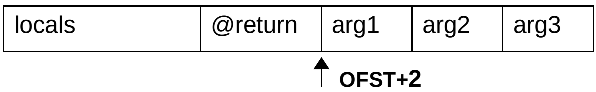
Stack Representation

The diagrams below show the stack layout at function entry *func*. In this example, *func* has three arguments: *arg1*, *arg2* and *arg3*. The first diagram describes cases where *arg1* is in the **d** register. The second diagram describes cases where *arg1* is not in the **d** register. In both cases, arguments are assumed to be widened, so char is widened to short and float to double.

arg1 is in d



arg1 not in d



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, *heap* and *stack* in the following way:

initialized variables (*data* segment)	uninitialized variables (*bss* segment)	*heap* growing upward and *stack* growing downward
	heap starts here ↑	*stack* starts here ↑

The heap start is the *bss* end, and is equal to the **__memory** symbol defined by the linker with an appropriate **+def** directive. The stack pointer is initialized by the application startup (*crts.s*) to an absolute value, generally the end of available memory, or a value relative to the end of the *bss* segment (for multi-tasking purposes for instance). The *heap* grows upwards and the *stack* downwards until collision may occur.

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 _memory;
      static char *_brk = NULL;/* memory break */
      char *obrk, yellow[40];

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

The *yellow* array is used to calculate the *stack* pointer value to check the *heap* limits. This array is declared as the **last** local variable, so its address is almost equal to the stack pointer once the function has been entered. It is declared to be 40 bytes wide to allow for some security margin. If the new top is outside the authorized limits, 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 separated memory area. The first example shown below handles the heap area 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 0x8000    # heap start
sbreak.o                # sbreak function
```

It is also possible to handle the heap area outside of any C object, just by defining the heap start and end values using the linker **+def** directives. Assuming these symbols are named **_heap_start** and **_heap_end** in C, it is possible to define them at link time with such directives:

```
+def __heap_start=0x8000# heap start
+def __heap_end=0xA000  # 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.*

You need to add an extra '_' character when defining a C symbol at link time to match the C compiler naming conventions.

The modified _*sbreak* function is as follows:

```
/*    SET SYSTEM BREAK IN MEMORY
 */
void *sbreak(int size)
      {
      extern char _heap_start, _heap_end;/* heap limits */
      static char *_brk = NULL;/* memory break */
      char *obrk;

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

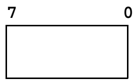
Note that it is possible to use this *_sbreak* function as a *malloc* equivalent function with some restrictions. The *malloc* function should be used when the allocated memory has to be released, or if the application has no idea about the total amount of space needed. If memory can be allocated and 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.h12                  # 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*.

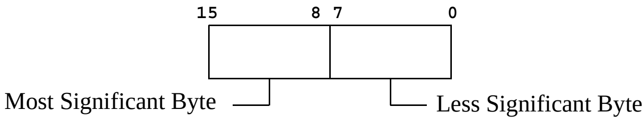
Data Representation

Data objects of type *char* are stored as one byte:



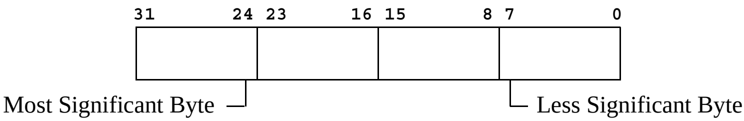
Char representation

Data objects of type *short int*, *int* and *16 bit pointers (@near)* are stored as two bytes, more significant byte first:



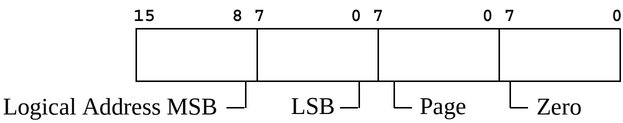
Short, Int, 16 bit Pointer

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



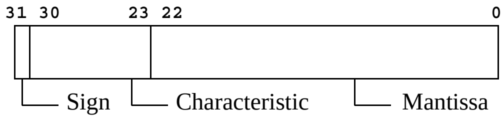
Long representation

Data objects of type *@far pointer* are stored as four bytes. The first word is the logical address represented as 16 bit pointer, the next byte is the paged value and the next byte is a 0.

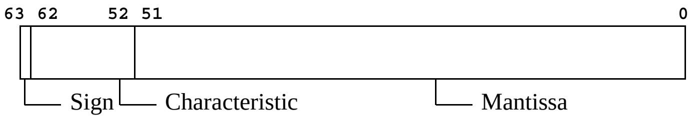


@far pointer representation

Data objects of type *float* and *double* are represented as for the proposed IEEE Floating Point Standard; four bytes (for float) or eight bytes (for double) 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 (for float) or eleven bits (for double) are the characteristic, biased such that the binary exponent of the number is the characteristic minus 126 (for float) or 1022 (for double); 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.



Float representation



Double representation

CHAPTER

4

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

To invoke the cross compiler, type the command **cx6812**, 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:

```
cx6812 [options] <files>.[c|s]
```

cx6812 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, **cx6812** 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:

```
cx6812 acia.c
```

compiles and assembles the *acia.c* C program, 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, **cx6812** 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 **cx6812** compiler accepts the following command line options, each of which is described in detail below:

```
  cx6812 [options] <files>
         -a*>   assembler options
         -ce*   path for errors files
         -cl*   path for listings files
         -co*   path for objects files
         -d*>   define symbol
         -ec    all C files
         -eg    eclipse error messages
         -ep    preprocess assembler files
         -es    all assembler files
         -ex    prefix executables
         -e     create error file
         -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
```

Cx6812 Option Usage

Option	Description
-a*>	specify assembler options. Up to 128 options can be specified on the same command line. See **Chapter 5**, “*Using The Assembler*”, for the list of all accepted options.
-ce*	specify a path for the error files. By default, errors are created in the same directory than the source files.
-cl*	specify a path for the listing files. By default, listings are created in the same directory than the source files.
-co*	specify a path for the object files. By default, objects are created in the same directory than the source files.
-d*>	specify * as the name of a user-defined preprocessor symbol (**#define**). The form of the definition is **-dsymbol[=value]**; the symbol is set to **1** if value is omitted. You can specify up to 128 such definitions.
-e	log errors from parser in a file instead of displaying them on the terminal screen. The error file name is defaulted to *<file>.err*, and is created only if there are errors.
-ec	treat all files as C source files.
-eg	produce error messages directly compatible with the Eclipse environment.
-ep	run the C preprocessor before assembling an assembler source fle.
-es	treat all files as assembler source files.
-ex	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.
-f*	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 *cx6812.cxf* in the compiler directory as specified in the installation process. For more information, see **Appendix B**, “*Modifying Compiler Operation*”.

Cx6812 Option Usage (cont.)

Option	Description
-g*>	specify code generation options. Up to 128 options can be specified. See **Appendix D**, "*The cg6812 Code Generator*", for the list of all accepted options.
-i*>	define include path. You can define up to 128 different paths. Each path is a directory name, **not** terminated by any directory separator character, or a file containing an unlimited list of directory names.
-l	merge C source listing with assembly language code; listing output defaults to *<file>.ls*.
-no	do not use the optimizer.
-o*>	specify optimizer options. Up to 128 options can be specified. See **Appendix D**, "*The co6812 Assembly Language Optimizer*", for the list of all accepted options.
-p*>	specify parser options. Up to 128 options can be specified. See **Appendix D**, "*The cp6812 Parser*", to get the list of all accepted options.
-s	create only assembler files and stop. Do not assemble the files produced.
-sm	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.
-sp	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.
-t*	specify path for temporary files. The path is a directory name, **not** terminated by any directory separator character.
-v	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.
-x	do not execute the passes, instead write to STDOUT the commands which otherwise would have been performed.

Cx6812 Option Usage (cont.)

Option	Description
+*>	select a predefined compiler option. These options are predefined 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
+ceven	split the constants into two sections, one for single byte constant (**.const)** and one for the other ones which are supposed to be accessed more efficiently when properly aligned (**.const.w**).
+debug	produce debug information to be used by the debug utilities provided with the compiler and by any external debugger.
+even	align any object larger than one byte on an even boundary.
+fast	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 "*Inlining Functions*" in **Chapter 3**.
+hcs	enable support for **HCS12** (**S12**).
+modf	force all functions to be compiled as **@far** functions. For more information, see "*Calling a Bank Switched Function*" in **Chapter 3**.
+nobss	do not use the **.bss** section. By default, such uninitialized variables are defined into the *.bss* section. This option is useful to force all variables to be grouped into a single section.
+nocst	output literals and contants in the code section **.text** instead of the specific section **.const**.
+nofds	do not use the **.fdata** section for variables allocated in paged memory. By default, such variables are defined into the *.fdata* section. This option is intended only for compatibility with previous versions.
+nofts	do not use the **.ftext** section for banked code. By default, banked code is allocated in the *.text* section. This option is intended only for compatibility with previous versions.

Cx6812 Option Usage (cont.)

Option	Description
+nowiden	do not widen char and float arguments. By default, char arguments are promoted to int before to be passed as argument.
+pgff	enable support for early **HC12DG128** family. This option is necessary only if no specific header file is included in order to locate the **DPAGE** register at its proper location.
+pic	produce position independant code. All function calls and const variables access are using an indexed **pc** relative addressing modes. The code can then be moved anywhere. This option enforces the **+nocst** option to map all the constants in the code space. For more information, see "*Using Position Independent Code*" in **Chapter 3**.
+picd	produce position independant code and data. All function calls and any variable access are using an indexed **pc** relative addressing modes. The code can then be moved anywhere. This option enforces the **+nocst** option to map all the constants in the code space, and maps all the data objects in the same code space. For more information, see "*Using Position Independent Code*" in **Chapter 3**.
+picds	produce position independant code and data. All function calls and any variable access are using an indexed **pc** relative addressing modes. The code can then be moved anywhere. This option enforces the **+nocst** option to map all the constants in the code space, but keeps all the data objects in their usual sections which should be linked contiguously to the code/const space. For more information, see "*Using Position Independent Code*" in **Chapter 3**.
+proto	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.
+rev	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 **+rev** option is specified, bitfields are filled from the *msb* to the *lsb*.

Cx6812 Option Usage (cont.)

Option	Description
+split	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 **-k** 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 "*Segment Control Options*" in **Chapter 6**.
+sprec	force all floating point arithmetic to single precision. If this option is enabled, all floats, doubles and long doubles are treated as float, and calculation are made in single precision. In such a case, the application must be linked with the **libf.h12** library instead of **libd.h12**.
+std	enable support for **HC12**. Default is **S12**.
+strict	enforce a stronger type checking. For more information, see "*Extra verifications*" in **Appendix D**.
+warn	enable warnings. For more information, see "*Warning Levels*" in **Appendix D**.
+zpage	force all data to be defined into the *.bsct* section. This option assumes that the full application declares less than the available space in the *.bsct* section. The linker should be configured to check the size. For more information, see "*Placing Data Objects in The Zero Page Section*" in **Chapter 3**.

Temporary Compiler files

By default, the compiler uses the **TMP** environment variable to locate its temporary files. You can override this behavior by using the **-t*** option. The compiler can also use the **CXTEMP** environment variable to specify the path for its temporary files. The path is a directory name, **not** terminated by any directory separator character. The compiler will then search the path specified by the **CXTEMP** environment variable. For example, setting the **CXTEMP** environment variable to the **C:\MyTemp** directory is done as follow:

```
C>set CXTEMP=C:\MyTemp
```


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
cp6812	<file>.c	<file>.1
cg6812	<file>.1	<file>.2
co6812	<file>.2	<file>.s
error listing	<file>.c	<file>.err
assembler listing	<file>.[c|s]	<file>.ls
C header files	<file>.h	
ca6812	<file>.s	<file>.o
source listing	<file>.s	<file>.ls
clnk	<file>.o	name required
cbank	<file>	STDOUT
chex	<file>	STDOUT
clabs	<file.h12>	<files>.la
clib	<file>	name required
cobj	<file>	STDOUT
cv695	<file>	<file>.695
cvdwarf	<file.h12>	<file.elf>

Generating Listings

You can generate listings of the output of any (or all) the compiler passes by specifying the **-l** option to **cx6812**. 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:

```
cx6812 -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 the **cx6812** compiler. You can locate the listing file in a different directory by using the **-ce** option. For example, you would type:

```
cx6812 -e prog.c
```

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

Return Status

cx6812 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:

```
cx6812 -v file.c
```

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

```
cx6812 -s file.c
```


C Library Support

This section describes the facilities provided by the C library. The C cross compiler for HC12/HCS12 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 four separate sub-libraries; one for machine-dependent routines (the *machine* library), one that does not support floating point (the *integer* library), one that provides full floating point support (the floating point library) and one that provides specific functions for fuzzy logic support (the *fuzzy* 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.

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.

Integer Library Functions

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

abs	isalnum	memchr	strchr
atoi	isalpha	memcmp	strcmp
atol	iscntrl	memcpy	strcpy
calloc	isdigit	memmove	strcspn
checksum	isgraph	memset	strlen
checksum16	islower	printf	strncat
checksum16f	isprint	putchar	strncmp
checksum16x	ispunct	puts	strncpy
checksum16xf	isqrt	rand	strpbrk
checksumf	isspace	realloc	strrchr

checksumx	isupper	sbreak	strspn
checksumxf	isxdigit	scanf	strstr
div	labs	setjmp	strtol
fctcpy	ldiv	sprintf	tolower
free	longjmp	srand	toupper
getchar	lsqrt	sscanf	vprintf
gets	malloc	strcat	vsprintf

Floating Point Library Functions

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

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

Fuzzy Library Functions

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

memhc12	revhc12	revwhc12	wavhc12

Common Input/Output Functions

Two of the functions that perform stream input/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

Three 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: *va_arg*, *va_end* and *va_start*.

Functions Implemented as Builtins

A few functions of the C library are actually implemented as “builtins”. The code for those functions is directly inlined instead of passing arguments and calling a function. Arguments are built directly in registers and the code is produced to match exactly the function behaviour. Those functions are also provided in the library to allow them to be called through pointers. The functions in the C library that are implemented as builtins are: *abs, max, min, memcpy, strcpy, strlen, strcmp, revhc12*, *revwhc12* and *wavhc12*.

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, iscntrl*, *isgraph*, *isprint*, *ispunct*, *isspace*, *isxdigit*, *isdigit*, *isupper, islower, tolower* and *toupper.*

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

<fuzzy.h> - Header file for fuzzy functions.

<io*.h> - Header files for input-output registers. Each register has an upper-case name which matches the standard NXP definition.. Header files specific to **HCS12** (**S12**) family members start with “`ios`”. Files specific to the standard **HC12** family members start with “`io`”. For processors allowing I/O registers to be moved, registers are mapped at a **base** address defaulted to **0x0**. If these registers are mapped at a different address, the preprocessor symbol **_BASE** must be defined with the expected location before including the specific header file.

<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, overflow*.

<setjmp.h> - Header file for nonlocal jumps: *setjmp* and *longjmp*.

<stdarg.h> - Header file for walking argument lists: *va_arg, va_end* and *va_start*. Use these macros with any function you write that must accept a variable number of arguments.

<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, calloc, div, exit, free, isqrt, labs, ldiv, lsqrt, malloc, rand, realloc, 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, since *int* is the function return type that the compiler assumed by default, 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

```
/* no header file need be included */
_asm(<string constant>, ...)
```

Function

_asm generates inline assembly code by copying <*string constant*> and quoting it into the output assembly code stream. If extra arguments are specified, they are processed as for a standard function. If arguments are stacked, they are popped off just after the inline code produced. For more information, see "*Inserting Inline Assembly Instructions*" in **Chapter 3**.

Return Value

Nothing, unless *_asm()* is used in an expression. In that case, normal return conventions must be followed. For more information, see "*Register Usage*" in **Chapter 3**.

Example

The sequence **tsx; pshx**, may be generated by the following call:

```
_asm("\ttsx\n\tpshx\n");
```

Notes

_asm() is not packaged in any library. It is recognized (and its argument passed unchanged) by the compiler itself.

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.

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 *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:

[+|-]d*[.d*][e[+|-]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 **l** 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 **l** 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.

calloc

Description

Allocate and clear space on the heap

Syntax

```
#include <stdlib.h>
void *calloc(int nelem, int elsize)
```

Function

calloc allocates space on the heap for an item of size *nbytes*, where *nbytes* = *nelem* * *elsize*. The space allocated is guaranteed to be at least *nbytes* long, starting from the pointer returned, which is guaranteed to be on a proper storage boundary for an object of any type. The heap is grown as necessary. If space is exhausted, *calloc* returns a null pointer. The pointer returned may be assigned to an object of any type without casting. The allocated space is initialized to zero.

Return Value

calloc returns a pointer to the start of the allocated cell if successful; otherwise it returns NULL.

Example

To allocate an array of ten doubles:

```
double *pd;
      pd = calloc(10, sizeof (double));
```

See Also

free, *malloc*, *realloc*

Notes

calloc 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 **b** register the value **0** or **1** depending on the *carry* bit value.

Return Value

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

Example

```
low <<= 1;          produces          lsl     _low
if (carry())                          bcc     L1
      ++high;                         inc     _high
                                L1:

low <<= 1;          produces          lsl     _low
high = carry()                        tfr     ccr,b
                                      andb    #1
                                      stab    _high
```

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 "*Inlining Functions*" 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.

_checkcrc16

Description

Verify the recorded checksum

Syntax

```
int _checkcrc16()
```

Function

_checkcrc16 scans the descriptor built by the linker and controls at the end that the computed CCITT 16 bit checksum is equal to the one expected. For more infomation, see “*Checksum Computation*” in **Chapter 6**.

Return Value

_checkcrc16 returns 0 if the checksum is correct, or a value different of 0 otherwise.

Example

```
if (_checkcrc16())
        abort();
```

Notes

The descriptor is built by the linker only if the *_checkcrc16* function is called by the application, even if there are segments marked with the **-ck** option.

_checkcrc16 is packaged in the integer library.

See Also

_checkcrc16x

NOTE

The routine provided by default does not support bank switching. Provision is made in the library source file to support bank switching if needed. Uncomment the proper PPAGE definition and re-assemble the file to link it with the application. It must be placed in the common area.

_checkcrc16x

Description

Verify the recorded checksum

Syntax

```
int _checkcrc16x()
```

Function

_checkcrc16x scans the descriptor built by the linker and controls at the end that the computed CCITT 16 bit checksum is equal to the one expected. For more infomation, see “*Checksum Computation*” in **Chapter 6**.

Return Value

_checkcrc16x returns 0 if all the checksums are correct, or a value different of 0 otherwise.

Example

```
if (_checkcrc16x())
        abort();
```

Notes

The descriptor is built by the linker only if the *_checkcrc16x* function is called by the application, even if there are segments marked with the **-ck** option.

_checkcrc16x is packaged in the integer library.

See Also

_checkcrc16

NOTE

The routine provided by default does not support bank switching. Provision is made in the library source file to support bank switching if needed. Uncomment the proper PPAGE definition and re-assemble the file to link it with the application. It must be placed in the common area.

_checksum

Description

Verify the recorded checksum

Syntax

```
int _checksum()
```

Function

_checksum 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

_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

NOTE

The routine provided by default does not support bank switching. Provision is made in the library source file to support bank switching if needed. Uncomment the proper PPAGE definition and re-assemble the file to link it with the application. It must be placed in the common area.

_checksumf

Description

Verify the recorded checksum (fast version)

Syntax

```
int _checksumf()
```

Function

_checksumf scans the descriptor built by the linker and controls at the end that the computed 8 bit checksum is equal to the one expected. *_checksumf* is the fast version of *_checksum*, costing a larger code space but getting the result in a shorter time. For more information, see "*Checksum Computation*" in **Chapter 6**.

Return Value

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

Example

```
if (_checksumf())
        abort();
```

Notes

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

_checksumf is packaged in the integer library.

See Also

_checksumxf, _checksum16f, _checksum16xf

NOTE

The routine provided by default does not support bank switching. Provision is made in the library source file to support bank switching if needed. Uncomment the proper PPAGE definition and re-assemble the file to link it with the application. It must be placed in the common area.

_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

NOTE

The routine provided by default does not support bank switching. Provision is made in the library source file to support bank switching if needed. Uncomment the proper PPAGE definition and re-assemble the file to link it with the application. It must be placed in the common area.

_checksumxf

Description

Verify the recorded checksum (fast version)

Syntax

```
int _checksumxf()
```

Function

_checksumxf scans the descriptor built by the linker and controls at the end that the computed 8 bit checksum is equal to the one expected. *_checksumxf* is the fast version of *_checksumx*, costing a larger code space but getting the result in a shorter time. For more information, see "*Checksum Computation*" in **Chapter 6**.

Return Value

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

Example

```
if (_checksumxf())
        abort();
```

Notes

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

_checksumxf is packaged in the integer library.

See Also

_checksumf, _checksum16f, _checksum16xf

NOTE

The routine provided by default does not support bank switching. Provision is made in the library source file to support bank switching if needed. Uncomment the proper PPAGE definition and re-assemble the file to link it with the application. It must be placed in the common area.

_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

NOTE

The routine provided by default does not support bank switching. Provision is made in the library source file to support bank switching if needed. Uncomment the proper PPAGE definition and re-assemble the file to link it with the application. It must be placed in the common area.

_checksum16f

Description

Verify the recorded checksum (fast version)

Syntax

```
int _checksum16f()
```

Function

_checksum16f scans the descriptor built by the linker and controls at the end that the computed 16 bit checksum is equal to the one expected. *_checksum16f* is the fast version of *_checksum16*, costing a larger code space but getting the result in a shorter time. For more information, see "*Checksum Computation*" in **Chapter 6**.

Return Value

_checksum16f returns 0 if the checksum is correct, or a value different of 0 otherwise.

Example

```
if (_checksum16f())
        abort();
```

Notes

The descriptor is built by the linker only if the *_checksum16f* function is called by the application, even if there are segments marked with the **-ck** option.

_checksum16f is packaged in the integer library.

See Also

_checksumf, _checksumxf, _checksum16xf

NOTE

The routine provided by default does not support bank switching. Provision is made in the library source file to support bank switching if needed. Uncomment the proper PPAGE definition and re-assemble the file to link it with the application. It must be placed in the common area.

_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

NOTE

The routine provided by default does not support bank switching. Provision is made in the library source file to support bank switching if needed. Uncomment the proper PPAGE definition and re-assemble the file to link it with the application. It must be placed in the common area.

_checksum16xf

Description

Verify the recorded checksum (fast version)

Syntax

```
int _checksum16xf()
```

Function

_checksum16xf scans the descriptor built by the linker and controls at the end that the computed 16 bit checksum is equal to the one expected. *_checksum16xf* is the fast version of *_checksum16x*, costing a larger code space but getting the result in a shorter time. For more information, see "*Checksum Computation*" in **Chapter 6**.

Return Value

_checksum16xf returns 0 if the checksum is correct, or a value different of 0 otherwise.

Example

```
if (_checksum16xf())
        abort();
```

Notes

The descriptor is built by the linker only if the *_checksum16xf* function is called by the application, even if there are segments marked with the **-ck** option.

_checksum16xf is packaged in the integer library.

See Also

_checksumf, _checksumxf, _checksum16f

NOTE

The routine provided by default does not support bank switching. Provision is made in the library source file to support bank switching if needed. Uncomment the proper PPAGE definition and re-assemble the file to link it with the application. It must be placed in the common area.

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 *nth* 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>
div_t div(int numer, int denom)
```

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.

eepcpy

Description

Copy a buffer to an **eeprom** buffer

Syntax

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

Function

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

Return Value

eepcpy returns s1.

Example

To place "*first string, second string*" in *eepbuf[]*:

```
eepcpy(eepbuf, "first string", 12);
eepcpy(eepbuf + 13, ", second string", 15);
```

See Also

eepset, *eepera*

Notes

eepcpy is packaged in the eeprom libraries: *liba* (for DG128A), *libb* (basic library) and *libe* (standard library). See "*Linking Library Objects*" in **Chapter 6** for more information.

eepera

Description
Erase the full *eeprom* space

Syntax
```
void eepera(void)
```

Function
eepera erases the full *eeprom* space with the global erase sequence. It does not erase the **config** register.

Return Value
Nothing.

Example
To erase the full *eeprom* space:

```
eepera();
```

See Also
eepset, eepcpy

Notes
eepera is packaged in the eeprom libraries: *liba* (for DG128A), *libb* (basic library) and *libe* (standard library). See "*Linking Library Objects*" in **Chapter 6** for more information..

eepset

Description

Propagate fill character throughout *eeprom* buffer

Syntax

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

Function

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

Return Value

eepset returns *s*.

Example

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

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

See Also

eepcpy, *eepera*

Notes

eepset is packaged in the eeprom libraries: *liba* (for DG128A), *libb* (basic library) and *libe* (standard library). See "*Linking Library Objects*" in **Chapter 6** for more information.

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 integer 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:

f = x - i * 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.

free

Description

Free space on the heap

Syntax

```
#include <stdlib.h>
void free(void *ptr)
```

Function

free returns an allocated cell to the heap for subsequence reuse. The cell pointer *ptr* must have been obtained by an earlier *calloc*, *malloc*, or *realloc* call; otherwise the heap will become corrupted. *free* does its best to check for invalid values of *ptr*. A NULL value for *ptr* is explicitly allowed, however, and is ignored.

Return Value

Nothing.

Example

To give back an allocated area:

```
free(pd);
```

See Also

calloc, *malloc*, *realloc*

Notes

No effort is made to lower the system break when storage is freed, so it is quite possible that earlier activity on the heap may cause problems later on the stack.

free is packaged in the integer 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.

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.

isalnum

Description

Test for alphabetic or numeric character

Syntax

```
#include <ctype.h>
int isalnum(int 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(int 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.

iscntrl

Description

Test for control character

Syntax

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

Function

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

Return Value

iscntrl 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 (iscntrl(*s))
                *s = '%';
```

See Also

isgraph, isprint, ispunct, isspace

Notes

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

iscntrl is packaged in the integer library.

isdigit

Description

Test for digit

Syntax

```
#include <ctype.h>
int isdigit(int 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(int 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

iscntrl, *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 lower-case character

Syntax

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

Function

islower tests whether *c* is a lower-case alphabetic character.

Return Value

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

Example

To convert to upper-case:

```
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(int 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

iscntrl, 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(int 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

iscntrl, *isgraph*, *isprint*, *isspace*

Notes

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

ispunct is packaged in the integer library.

isqrt

Description

Integer square root

Syntax

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

Function

isqrt obtains the integral square root of the unsigned int *i*.

Return Value

isqrt returns the closest integer smaller or equal to the square root of *i*, expressed as an **unsigned int**.

Example

To use *isqrt* to check whether *n > 2* is a prime number:

```
if (!(n & 01))
        return (NOTPRIME);
sq = isqrt(n);
for (div = 3; div <= sq; div += 2)
        if (!(n % div))
                return (NOTPRIME);
return (PRIME);
```

See Also

lsqrt, sqrt

Notes

isqrt is packaged in the integer library.

isspace

Description

Test for whitespace character

Syntax

```
#include <ctype.h>
int isspace(int 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

iscntrl, 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 upper-case character

Syntax

```
/* no header file need be included */
int isupper(int c)
```

Function

isupper tests whether *c* is an upper-case alphabetic character.

Return Value

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

Example

To convert to lower-case:

```
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(int 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 << 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.

longjmp

Description

Restore calling environment

Syntax

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val)
```

Function

longjmp restores the environment saved in *env* by *setjmp*. If *env* has not been set by a call to *setjmp*, or if the caller has returned in the meantime, the resulting behavior is unpredictable.

All accessible objects have their values restored when *longjmp* is called, except for objects of storage class register, the values of which have been changed between the *setjmp* and *longjmp* calls.

Return Value

When *longjmp* returns, program execution continues as if the corresponding call to *setjmp* had returned the value *val*. *longjmp* cannot force *setjmp* to return the value zero. If *val* is zero, *setjmp* returns the value one.

Example

You can write a generic error handler as:

```
void handle(int err)
      {
      extern jmp_buf env;
      longjmp(env, err); /* return from setjmp */
      }
```

See Also

setjmp

Notes

longjmp is packaged in the integer library.

lsqrt

Description

Long integer square root

Syntax

```
#include <stdlib.h>
unsigned int lsqrt(unsigned long l)
```

Function

lsqrt obtains the integral square root of the unsigned long *l*.

Return Value

lsqrt returns the closest integer smaller or equal to the square root of *l*, expressed as an **unsigned int**.

Example

To use *lsqrt* to check whether *n > 2* is a prime number:

```
if (!(n & 01))
        return (NOTPRIME);
sq = lsqrt(n);
for (div = 3; div <= sq; div += 2)
        if (!(n % div))
                return (NOTPRIME);
return (PRIME);
```

See Also

isqrt, sqrt

Notes

lsqrt is packaged in the integer library.

malloc

Description

Allocate space on the heap

Syntax

```
#include <stdlib.h>
void *malloc(unsigned int nbytes)
```

Function

malloc allocates space on the heap for an item of size *nbytes*. The space allocated is guaranteed to be at least *nbytes* long, starting from the pointer returned, which is guaranteed to be on a proper storage boundary for an object of any type. The heap is grown as necessary. If space is exhausted, *malloc* returns a null pointer.

Return Value

malloc returns a pointer to the start of the allocated cell if successful; otherwise it returns NULL. The pointer returned may be assigned to an object of any type without casting.

Example

To allocate an array of ten doubles:

```
double *pd;
      pd = malloc(10 * sizeof *pd);
```

See Also

calloc, free, realloc

Notes

malloc is packaged in the integer 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 *builtin* function, 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 builtin declared in the *<stdlib.h>* header file. You can use it by including *<stdlib.h>* with your program. Because it is a builtin, *max* cannot be called from non-C programs, nor can its address be taken.

memchr

Description

Scan buffer for character

Syntax

```
#include <string.h>
void *memchr(void *s, int c, unsigned int 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 int 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 int 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 implemented as a *builtin* function.

memhc12

Description

Fuzzify an input

Syntax

```
#include <fuzzy.h>
void memhc12(char crisp, char *t_shape,
                  char *t_result, int nba)
```

Function

memchc12 evaluates the grade of membership of all the adjectives associated to an input. Each adjective is described by a shape and an output byte. The input value is specified by the **crisp** argument, and is followed by a pointer **t_shape** to an array of shapes (four bytes each), and a pointer **t_result** to an array of output addresses. The last argument **nba** specifies the number of output to evaluate.

Return Value

memhc12 sets the content of the array of *nba* bytes specified by *t_shape* to the grade of membership of the input *crisp* according to an array of shapes specified by *t_result*.

Example

```
memhc12(val, tabsh, tabptr, 4);
```

See Also

revhc12, revwhc12, wavhc12

memmove

Description

Copy one buffer to another

Syntax

```
#include <string.h>
void *memmove(void *s1, void *s2, unsigned int 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, int c, unsigned int 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.

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 *builtin* function, 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 builtin declared in the *<stdlib.h>* header file. You can use it by including *<stdlib.h>* with your program. Because it is a builtin, *min* cannot be called from non-C programs, nor can its address be taken.

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.

overflow

Description

Test or get the carry bit

Syntax

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

Function

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

Return Value

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

Example

```
low <<= 1;          produces            lsl     _low
if (overflow())                         bvc     L1
      ++high;                           inc     _high
                                  L1:

low <<= 1;          produces            lsl     _low
high = overflow()                       tfr     ccr,b
                                        andb    #2
                                        lsrb
                                        stab    _high
```

Notes

overflow 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 "*Inlining Functions*" in **Chapter 3**.

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(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 least significant byte of the int 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 - the argument is taken to be a *char* * pointer to a string. 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(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.

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.

realloc

Description

Reallocate space on the heap

Syntax

```
#include <stdlib.h>
void *realloc(void *ptr, unsigned int nbytes)
```

Function

realloc grows or shrinks the size of the cell pointed to by *ptr* to the size specified by *nbytes*. The contents of the cell will be unchanged up to the lesser of the new and old sizes. The cell pointer *ptr* must have been obtained by an earlier *calloc*, *malloc*, or *realloc* call; otherwise the heap will become corrupted.

Return Value

realloc returns a pointer to the start of the possibly moved cell if successful. Otherwise *realloc* returns NULL and the cell and *ptr* are unchanged. The pointer returned may be assigned to an object of any type without casting.

Example

To adjust *p* to be *n* doubles in size:

```
p = realloc(p, n * sizeof(double));
```

See Also

calloc, *free*, *malloc*

Notes

realloc is packaged in the integer library.

revhc12

Description

Evaluate fuzzy outputs

Syntax

```
#include <fuzzy.h>
void revhc12(char *rules, char *in_out)
```

Function

revhc12 evaluates fuzzy outputs based on evaluation rules which are specified by the pointer **rules**. The base address of input and output bytes is specified by the pointer **in_out**. Refer to the "*HC12/HCS12 Reference Manual*", for a complete description of the rules encoding.

Return Value

revhc12 sets the output bytes to the result values according to the evaluation rules.

Example

```
revhc12(rules, base);
```

See Also

memhc12, *revwhc12*, *wavhc12*

Notes

revhc12 is a builtin function declared in the <*fuzzy.h*> header file.

revwhc12

Description

Evaluate fuzzy outputs

Syntax

```
#include <fuzzy.h>
void revwhc12(unsigned int *rules, char *weight)
```

Function

revwhc12 evaluates fuzzy outputs based on evaluation rules which are specified by the pointer **rules**. If the **weight** pointer is provided, it should point to an array of byte weights used to ponderate the evaluation. Otherwise *weight* has to be specified has the NULL pointer. Refer to the “*HC12/HCS12 Reference Manual*”, for a complete description of the rules encoding.

Return Value

revwhc12 sets the output bytes to the result values according to the evaluation rules.

Example

```
revwhc12(rules, NULL);
```

See Also

memhc12, revhc12, wavhc12

Notes

revwhc12 is a builtin function declared in the <*fuzzy.h*> header file.

sbreak

Description

Allocate new memory

Syntax

```
/* no header file need be included */
void *sbreak(unsigned int size)
```

Function

sbreak modifies the program memory allocation as necessary, to make available at least *size* contiguous bytes of new memory, on a storage boundary adequate for representing any type of data. There is no guarantee that successive calls to *sbreak* will deliver contiguous areas of memory.

Return Value

sbreak returns a pointer to the start of the new memory if successful; otherwise the value returned is NULL.

Example

To buy space for an array of symbols:

```
if (!(p = sbreak(nsyms * sizeof (symbol))))
        remark("not enough memory!", NULL);
```

Notes

sbreak is packaged in the integer library.

sbreak is an extension to the ANSI C standard.

scanf

Description

Read formatted input

Syntax

```
#include <stdio.h>
int scanf(char *fmt,...)
```

Function

scanf reads formatted input from the output stream using the format string at *fmt* and the arguments specified by ..., as described below.

scanf uses *getchar* to read each character.

The behavior is unpredictable if there are insufficient argument pointers for the format. If the format string is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored.

Format Specifiers

The format string may contain:

- any number of spaces, horizontal tabs, and newline characters which cause input to be read up to the next non-whitespace character, and
- ordinary characters other than '**%**' which must match the next character of the input stream.

Each <*conversion specification*>, the definition of which follows, consists of the character '**%**', an optional assignment-suppressing character '*****', an optional maximum field width, an optional **h**, **l** or **L** indicating the size of the receiving object, and a <*conversion character*>, described below.

A conversion specification directs the conversion of the next input field. The result is placed in the object pointed to by the subsequent argument, unless assignment suppression was indicated by a '*****'. An input field is a string of non-space characters; it extends to the next conflicting character or until the field width, if specified, is exhausted.

The conversion specification indicates the interpretation of the input field; the corresponding pointer argument must be a restricted type. The <*conversion character*> is one of the following:

% - a single **%** is expected in the input at this point; no assignment occurs.

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

c - a character is expected; the subsequent argument must be of type pointer to char. The normal behavior (skip over space characters) is suppressed in this case; to read the next non-space character, use **%1s**. If a field width is specified, the corresponding argument must refer to a character array; the indicated number of characters is read.

d - a decimal integer is expected; the subsequent argument must be a pointer to integer.

e, f, g - a float is expected; the subsequent argument must be a pointer to float. The input format for floating point numbers is an optionally signed sequence of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an **E** or **e**, followed by an optionally signed integer.

i - an integer is expected; the subsequent argument must be a pointer to integer. If the input field begins with the characters **0x** or **0X**, the field is taken as a hexadecimal integer. If the input field begins with the character 0, the field is taken as an octal integer. Otherwise, the input field is taken as a decimal integer.

n - no input is consumed; the subsequent argument must be an *int ** pointer to an integer into which is written the number of characters read from the input stream so far by this call to *scanf*.

o - an octal integer is expected; the subsequent argument must be a pointer to integer.

p - a pointer is expected; the subsequent argument must be a *void ** pointer. The format of the input field should be the same as that produced by the **%p** conversion of *printf*. On any input other than a value printed earlier during the same program execution, the behavior of the **%p** conversion is undefined.

s - a character string is expected; the subsequent argument must be a *char* * pointer to an array large enough to hold the string and a terminating NUL, which will be added automatically. The input field is terminated by a space, a horizontal tab, or a newline, which is not part of the field.

u - an unsigned decimal integer is expected; the subsequent argument must be a pointer to integer.

x - a hexadecimal integer is expected; a subsequent argument must be a pointer to integer.

[- a string that is not to be delimited by spaces is expected; the subsequent argument must be a *char* * just as for **%s**. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not a circumflex '^', the input field consists of all characters up to the first character that is not in the set between the brackets; if the first character after the left bracket is a circumflex, the input field consists of all characters up to the first character that is in the set of the remaining characters between the brackets. A NUL character will be appended to the input.

The conversion characters **d**, **i**, **o**, **u** and **x** may be preceded by **l** to indicate that the subsequent argument is a pointer to long int rather than a pointer to int, or by **h** to indicate that it is a pointer to short int. Similarly, the conversion characters **e** and **f** may be preceded by **l** to indicate that the subsequent argument is a pointer to double rather than a pointer to float, or by **L** to indicate a pointer to long double.

The conversion characters **e**, **g** or **x** may be capitalized. However, the use of upper case has no effect on the conversion process and both upper and lower case input is accepted.

If conversion terminates on a conflicting input character, that character is left unread in the input stream. Trailing white space (including a newline) is left unread unless matched in the control string. The success of literal matches and suppressed assignments is not directly determinable other than via the **%n** conversion.

Return Value

scanf returns the number of assigned input items, which can be zero if there is an early conflict between an input character and the format, or EOF if end of file is encountered before the first conflict or conversion.

Example

To be certain of a dubious request:

```
printf("are you sure?");
if (scanf("%c", &ans) && (ans == 'Y' || ans == 'y'))
        scrog();
```

See Also

sscanf

Notes

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

If an invalid conversion specifier is encountered, it is ignored.

setjmp

Description

Save calling environment

Syntax

```
#include <setjmp.h>
int setjmp(jmp_buf env)
```

Function

setjmp saves the calling environment in *env* for later use by the *longjmp* function.

Since *setjmp* manipulates the stack, it should never be used except as the single operand in a switch statement.

Return Value

setjmp returns zero on its initial call, or the argument to a *longjmp* call that uses the same *env*.

Example

To call any event until it returns 0 or 1 and calls *longjmp*, which will then start execution at the function *event0* or *event1*:

```
static jmp_buf ev[2];

      switch (setjmp(ev[0]))
             {
      case 0:             /* registered */
             break;
      default:            /* event 0 occurred */
             event0();
             next();
             }
      switch (setjmp(ev[1])
             {
      case 0:             /* registered */
             break;
      default:            /* event 1 occurred */
             event1();
             next();
```



```
                }
        next();
                ...
next()
        {
        int i;

        for (; ;)
                {
                i = anyevent();
                if (i == 0 || i == 1)
                        longjmp(ev[i]);
                }
        }
```

See Also

longjmp

Notes

setjmp 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, 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);
```

See Also

isqrt, lsqrt

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 0.

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.

sscanf

Description

Read formatted input from a string

Syntax

```
#include <stdio.h>
int sscanf(schar *, char *fmt,...)
```

Function

sscanf reads formatted input from the NUL-terminated string pointed at by **s** using the format string at *fmt* and the arguments specified by ..., in exactly the same way as *scanf*. See the description of the *scanf* function for information on the format conversion specifiers.

Return Value

sscanf returns the number of assigned input items, which can be zero if there is an early conflict between an input character and the format, or EOF if the end of the string is encountered before the first conflict or conversion.

See Also

scanf

Notes

sscanf is packaged in both the integer library and the floating point library. The functionality of the integer only version of *sscanf* 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 *sscanf*, the following conversion specifiers are invalid: **e**, **f**, **g** and **p**. The **L** flag is also invalid.

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 *s, int 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, strrchr, 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 implemented as a *builtin* 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.

strncat

Description

Concatenate strings of length n

Syntax

```
#include <string.h>
char *strncat(char *s1, char *s2, unsigned int 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 int 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 int 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.

strrchr

Description

Scan string for last occurrence of character

Syntax

```
#include <string.h>
char *strrchr(char *s,int 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

strrchr 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, strrchr, 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, int 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 *l*:

```
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,
                      int 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 lower-case if necessary

Syntax

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

Function

tolower converts an upper-case letter to its lower-case equivalent, leaving all other characters unmodified.

Return Value

tolower returns the corresponding lower-case 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 upper-case if necessary

Syntax

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

Function

toupper converts a lower-case letter to its upper-case equivalent, leaving all other characters unmodified.

Return Value

toupper returns the corresponding upper-case letter, or the unchanged character.

Example

To convert a character string to upper-case letters:

```
for (i = 0; i < size; ++i)
        buf[i] = toupper(buf[i]);
```

See Also

tolower

Notes

toupper is packaged in the integer library.

va_arg

Description

Get pointer to next argument in list

Syntax

```
#include <stdarg.h>
type va_arg(va_list ap, type)
```

Function

The macro **va_arg** is an *rvalue* that computes the value of the next argument in a variable length argument list. Information on the argument list is stored in the array data object *ap*. You must first initialize *ap* with the macro *va_start*, and compute all earlier arguments in the list by expanding *va_arg* for each argument.

The type of the next argument is given by the type name *type*. The type name must be the same as the type of the next argument. Remember that the compiler widens an arithmetic argument to int, and converts an argument of type float to double. You write the type after conversion. Write *int* instead of *char* and *double* instead of *float*.

Do not write a type name that contains any parentheses. Use a type definition, if necessary, as in:

```
typedef int (*pfi)();
      /* pointer to function returning int */
      ...
fun_ptr = va_arg(ap, pfi);
      /* get function pointer argument */
```

Return Value

va_arg expands to an rvalue of type *type*. Its value is the value of the next argument. It alters the information stored in *ap* so that the next expansion of *va_arg* accesses the argument following.

Example

To write multiple strings to a file:

```
#include <stdio.h>
#include <stdarg.h>

main()
      {
      void strput();
      strput(pf, "This is one string\n", \
             "and this is another...\n", (char *)0);
      }

void strput(FILE *pf,...);
void strput(char *ptr,...)
void strput(ptr)
      char *ptr;
      {
      char ptr;
      va_list va;

      if (!ptr)
             return;
      else
             {
             puts(ptr);
             va_start(va, ptr);
             while (ptr = va_arg(va, char *)
                    puts(ptr);
             va_end(va);
             }
      }
```

See Also

va_end, *va_start*

Notes

va_arg is a macro declared in the <*stdarg.h*> header file. You can use it with any function that accepts a variable number of arguments, by including <*stdarg.h*> with your program.

va_end

Description

Stop accessing values in an argument list

Syntax

```
#include <stdarg.h>
void va_end(va_list ap)
```

Function

va_end is a macro which you must expand if you expand the macro *va_start* within a function that contains a variable length argument list. Information on the argument list is stored in the data object designated by *ap*. Designate the same data object in both *va_start* and *va_end*.

You expand *va_end* after you have accessed all argument values with the macro *va_arg*, before your program returns from the function that contains the variable length argument list. After you expand *va_end*, do not expand *va_arg* with the same *ap*. You need not expand *va_arg* within the function that contains the variable length argument list.

You must write an expansion of *va_end* as an expression statement containing a function call. The call must be followed by a semicolon.

Return Value

Nothing. *va_end* expands to a statement, not an expression.

Example

To write multiple strings to a file:

```
#include <stdio.h>
#include <stdarg.h>

main()
      {
      void strput();

      strput(pf, “This is one string\n”, \
             “and this is another...\n”, (char *)0);
      }
```



```
void strput(FILE *pf,...);
void strput(char *ptr,...)
void strput(ptr)
      char *ptr;
      {
      char ptr;
      va_list va;

      if (!ptr)
             return;
      else
             {
             puts(ptr);
             va_start(va, ptr);
             while (ptr = va_arg(va, char *)
                    puts(ptr);
             va_end(va);
             }
      }
```

See Also

va_arg, *va_start*

Notes

va_end is a macro declared in the <*stdarg.h*> header file. You can use it with any function that accepts a variable number of arguments, by including <*stdarg.h*> with your program.

va_start

Description

Start accessing values in an argument list

Syntax

```
#include <stdarg.h>
void va_start(va_list ap, parmN)
```

Function

va_start is a macro which you must expand before you expand the macro *va_arg*. It initializes the information stored in the data object designated by *ap*. The argument *parmN* must be the identifier you declare as the name of the last specified argument in the variable length argument list for the function. In the function prototype for the function, *parmN* is the argument name you write just before the ,...

The type of *parmN* must be one of the types assumed by an argument passed in the absence of a prototype. Its type must not be float or char. Also, *parmN* cannot have storage class register.

If you expand *va_start*, you must expand the macro *va_end* before your program returns from the function containing the variable length argument list.

You must write an expansion of *va_start* as an expression statement containing a function call. The call must be followed by a semicolon.

Return Value

Nothing. *va_start* expands to a statement, not an expression.

Example

To write multiple strings to a file:

```
#include <stdio.h>
#include <stdarg.h>

main()
      {
```

```
        void strput();
        strput(pf, "This is one string\n", \
                "and this is another...\n", (char *)0);
        }

void strput(FILE *pf,...);
void strput(char *ptr,...)
void strput(ptr)
        char *ptr;
        {
        char ptr;
        va_list va;

        if (!ptr)
                return;
        else
                {
                puts(ptr);
                va_start(va, ptr);
                while (ptr = va_arg(va, char *)
                        puts(ptr);
                va_end(va);
                }
        }
```

See Also

va_arg, va_end

Notes

va_start is a macro declared in the <*stdarg.h*> header file. You can use it with any function that accepts a variable number of arguments, by including <*stdarg.h*> with your program.

vprintf

Description

Output arguments formatted to stdout

Syntax

```
#include <stdio.h>
#include <stdarg.h>
int vprintf(char *s, char fmt, va_list ap)
```

Function

vprintf writes formatted to the output stream using the format string at *fmt* and the arguments specified by pointer *ap*, in exactly the same way as *printf*. See the description of the *printf* function for information on the format conversion specifiers. The *va_start* macro must be executed before to call the *vprintf* function.

vprintf uses *putchar* to output each character.

Return Value

vprintf returns the numbers of characters transmitted.

Example

To format a double at *d* into *buf*:

```
va_start(aptr, fmt);
vprintf(fmt, aptr);
```

See Also

printf, *vsprintf*

Notes

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

vsprintf

Description

Output arguments formatted to buffer

Syntax

```
#include <stdio.h>
#include <stdarg.h>
int vsprintf(char *s, char fmt, va_list ap)
```

Function

vsprintf writes formatted to the buffer pointed at by **s** using the format string at *fmt* and the arguments specified by pointer *ap*, 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. The *va_start* macro must be executed before to call the *vsprintf* function.

Return Value

vsprintf returns the numbers of characters written, not including the terminating NUL character.

Example

To format a double at *d* into *buf*:

```
va_start(aptr, fmt);
vsprintf(buf, fmt, aptr);
```

See Also

printf, vprintf

Notes

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

wavhc12

Description

Evaluate weighted average

Syntax

```
#include <fuzzy.h>
char wavhc12(char *op1, char *op2, int nbval)
```

Function

wavhc12 computes the average of the a list of products between two arrays provided by the arguments **op1** and **op2**. The argument **nbval** specifies the number of input values to be used. Refer to the “*HC12/HCS12 Reference Manual*”, for a complete description of the calculation.

Return Value

wavhc12 returns the average in the **b** register.

Example

```
res = wavhc12(tab_1, tab_2, 10);
```

See Also

memhc12, revhc12, revwhc12

Notes

wavhc12 is a builtin function declared in the <*fuzzy.h*> header file.

CHAPTER

5

Using The Assembler

The **ca6812** cross assembler translates your assembly language source files into relocatable object files. The C cross compiler calls *ca6812* to assemble your code automatically, unless specified otherwise. *ca6812* generates also listings if requested. This chapter includes the following sections:

Invoking ca6812

ca6812 accepts the following command line options, each of which is described in detail below:

```
  ca6812 [options] <files>
        -a    absolute assembler
        -b    do not optimizes 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 a listing
        +l*   listing file name
        -m    accept old syntax
        -md   make dependencies
        -mi   accept label syntax
        -n*   processor name
        -o*   output file name
        -pe   all equates public
        -pl   keep local symbol
        -p    all symbols public
        -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
```

Ca6812 Option Usage

Option	Description
-a	map all sections to absolute, including the predefined ones.

Ca6812 Option Usage (cont.)

Option	Description
-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.
-c	produce cross-reference information. The cross-reference information will be added at the end of the listing file. This option enforces the **-l** option.
-d*>	where ***** has the form **name=value**, defines **name** to have the value specified by **value**. This option is equivalent to using an **equ** directive in each of the source files.
+e*	log errors from assembler in the text file ***** instead of displaying the messages on the terminal screen.
-ff	use *formfeed* character to skip pages in listing instead of using blank lines.
-ft	output a title in listing (date, file name, page). By default, no title is output.
-f#	define the value of the filling byte used to fill any gap created by the assembler directives. Default is **0**.
-ge	produce error messages directly compatible with the Eclipse environment
-h*	include the file specified by ***** before starting assembly. It is equivalent to an **include** directive in each source file.
-i*>	define a path to be used by the **include** directive. Up to 128 paths can be defined. A path is a directory name and is **not** ended by any directory separator character, or a file containing an unlimited list of directory names.
-l	create a listing file. The name of the listing file is derived from the input file name by replacing the suffix by the '*.ls*' extension, unless the **+l** option has been specified.
+l*	create a listing file in the text file *. If both **-l** and **+l** are specified, the listing file name is given by the **+l** option.

Ca6812 Option Usage (cont.)

Option	Description
-m	accept the old NXP syntax.
-md	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.
-mi	accept label that is not ended with a '**:**' character.
-n*	select the processor type. The default type is the HCS12 processor. The allowed targets are: **a** Standard processors (HC12) **s** **HCS12** family.
-o*	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 right most extension in the input file name with the character '**o**'. For example, if the input file name is prog.s, the default output file name is *prog.o*.
-pe	mark all symbols defined by an **equ** directive as **public**. This option has the same effect than adding a **xdef** directive for each of those symbols.
-pl	put locals in the symbol table. They are not published as externals and will be only displayed in the linker map file.
-p	mark all defined symbols as **public**. This option has the same effect than adding a **xdef** directive for each label.
-si	suppress the **.info.** section produced automatically and containing the object name, date and assembler options.
-u	produce an error message in the listing file for all occurrence of an undefined symbol. This option enforces the **-l** option.
-v	display the name of each file which is processed.
-x	add line debug information to the object file.
-xp	do not prefix filenames in the debug information with any absolute path name. Debuggers will have to be informed about the actual files location.

Ca6812 Option Usage (cont.)

Option	Description
-xx	add debug information in the object file for any label defining code or data. This option disables the **-p** 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, *ca6812* generates an object file. If requested by the **-l** or **-c** options, *ca6812* 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.

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 *ca6812* conforms to the NXP 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

ca6812 recognizes the following instructions:

```
aba     bpl     ediv    lbls    neg     sev
abx     bra     edivs   lblt    nega    sex
aby     brclr   emacs   lbmi    negb    staa
adca    brn     emaxd   lbne    nop     stab
adcb    brset   emaxm   lbpl    oraa    std
adda    bset    emind   lbra    orab    stop
addb    bsr     eminm   lbrn    orcc    sts
addd    bvc     emul    lbsr    psha    stx
anda    bvs     emuls   lbvc    pshb    sty
andb    call    eora    lbvs    pshc    suba
andcc   cba     eorb    ldaa    pshd    subb
asl     clc     etbl    ldab    pshx    subd
asla    cli     exg     ldd     pshy    swi
aslb    clr     fdiv    lds     pula    tab
asld    clra    ibeq    ldx     pulb    tap
asr     clrb    ibne    ldy     pulc    tba
asra    clv     idiv    leas    puld    tbeq
asrb    cmpa    idivs   leax    pulx    tbl
bcc     cmpb    inc     leay    puly    tbne
bclr    com     inca    lsl     rev     tfr
bcs     coma    incb    lsla    revw    tpa
beq     comb    ins     lslb    rol     tst
bge     cpd     inx     lsld    rola    tsta
bgnd    cps     iny     lsr     rolb    tstb
bgt     cpx     jmp     lsra    ror     tsx
bhi     cpy     jsr     lsrb    rora    tsy
bhs     daa     lbcc    lsrd    rorb    txs
bita    dbeq    lbcs    maxa    rtc     tys
bitb    dbne    lbeq    maxm    rti     wai
ble     dec     lbge    mem     rts     wav
blo     deca    lbgt    mina    sba     xgdx
bls     decb    lbhi    minm    sbca    xgdy
blt     des     lbhs    movb    sbcb
bmi     dex     lble    movw    sec
bne     dey     lblo    mul     sei
```

The **operand** field of an instruction uses an addressing mode to describe the instruction argument. The following example demonstrates the accepted syntax:

```
tpa                           ; implicit
ldaa    #1                    ; immediate
anda    var                   ; direct or extended
```

```
addd     ,x                ; indexed
orab     0,x               ; indexed
ldd      1,y+              ; indexed
jmp      [d,pc]            ; indexed
bne      loop              ; relative
bset     var,2             ; bit number
brset    var,2,loop        ; bit test and branch
```

The assembler chooses the smallest addressing mode where several solutions are possible. *Direct* addressing mode is selected when using a label defined in the *.bsct* section.

The assembler accepts **pc relative** addressing mode with two possible syntaxes:

```
ldd      10,pc             ; absolute offset
ldd      symbol,pcr        ; relative offset
```

The first syntax using the register name **pc** encoded the specified offset directly in the instruction. The second syntax using the register name **pcr** encodes in the instruction a relative value computed by substracting the value of the current *pc* from the value of the specific offset. This is mainly used with symbolic references.

Wherever the extended addressing mode is not accepted, the assembler will automatically replace it by an indexed addressing mode using the **pcr** relative notation if accepted by the instruction. Then, the two following lines produce the same code:

```
ldd      [symbol,pcr]
ldd      [symbol]          ; implied pcr
```

For an exact description of the above instructions, refer to the NXP's *HC12/HCS12 Reference 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*', '*b*', '*x*' and '*y*' are reserved and cannot be used as labels.


```
data1: dc.b     $56
_reg:  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:brset    1,PORTA,\@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$:    deca
       bne      1$
2$:    decb
       bne      2$
```

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$:   ldab     var
       beq      10$
       stab     var2
       local
10$:   ldaa     var2
       beq      10$
       staa     var
       rts
function2:
```

```
10$:    ldaa    var2
        suba    var
        bne     10$
        rts
```


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
10	decimal (no prefix)
%1010	binary
@12	octal
$1F or 0x1F or 0X1F	hexadecimal

The assembler also accepts numerics constants in different bases depending on a *suffix* character as follow:

Suffix	Base
D, d or none	decimal (no prefix)
B or b	binary
Q or q	octal
0AH or 0Ah	hexadecimal

The suffix letter can be entered uppercase 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:  cmp      #'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 **substraction** 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.

defined(symbol)	symbol defined

This special operator is evaluated as **1** if the **symbol** is defined by either a **equ** or **set** directive, or a **-d** option from the assembler command line. It is evaluated as 0 otherwise.

high(expression)	upper byte
low(expression)	lower byte
page(expression)	page byte
phigh(expression)	physical upper byte/word
plow(expression)	physical lower word

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.

phigh - extract the upper byte or word of the physical value of a symbolic expression

plow - extract the lower word of the physical value of a symbolic expression

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
       clra               ; prepare MSB
       tstb               ; test sign
       bpl      \@pos     ; if not positive
       coma               ; 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:

```
       ldab     char+1    ; load LSB
       signex             ; expand macro
       std      char      ; store result
```

The resulting code will be the same as if the following code had been written:

```
       ldab     char+1    ; load LSB
       clra               ; prepare MSB
       tstb               ; test sign
       bpl      pos       ; if not positive
       coma               ; invert MSB
pos:
       std      char      ; 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 catenate 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
       clra                ; prepare MSB
       ldab     \1         ; load LSB
       bpl      \@pos      ; if not positive
       coma                ; invert MSB
\@pos: std      \1         ; 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
       clra                        ; prepare MSB
       ldab     \value             ; load LSB
       bpl      \@pos              ; if not positive
       coma                        ; invert MSB
\@pos:  std     \(value)           ; store MSB
       endm                        ; end of macro
```

The form of a macro call is:

```
name>[.<ext>]  [<arguments>]
```

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. 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
ldx     #message
jsr     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                     ; otherwise
inx                      ; increment X register
endif
```

If the symbol *offset* is not one, the macro `addptr` is expanded with *offset* as argument, otherwise the *inx* 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.

The assembler predefines the symbol **__CSMC__** equal to value **1**. This symbol can be tested either with a conditional directive directly or by using the **defined()** special operator (see *Expressions* paragraph).

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: ldx      #value  ; fills text section
       jmp      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
.text	executable code
.data	initialized data
.bss	uninitialized data
.bsct	initialized data in zero page
.ubsct	uninitialized data in zero page

The sections **.bsct** and **.ubsct** are used for locating data in the *zero* page of the processor. The *zero* page 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 zero page 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.

```
       xref     var
       xref.b   zvar
       switch   .bsct
zvar2: ds.b     1
       switch   .bss
var2:  ds.b     1
       switch   .text
       ldaa     var
       ldaa     zvar
       ldaa     var2
       ldaa     var2
       end
```

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.

```
beq     farlabel  becomes     bne     *+5
                              jmp     farlabel
```

Note that a **`bra`** instruction will be replaced by a single **`jmp`** instruction if it cannot be encoded as a relative branch.

A **`jmp`** or **`jsr`** instruction will be replaced by a **`bra`** or **`bsr`** 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 *ca6812* 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
lda     #377    ; load $FF
```

bsct

Description

Switch to the predefined **.bsct** section.

Syntax

```
bsct
```

Function

The **bsct** directive switches input to a section named **.bsct**, also known as the **zero page** section. The assembler will automatically select the direct addressing mode when referencing an object defined in the *.bsct* section.

Example

```
        bsct
c_reg:
        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 **fcb** is alias to **dc.b**, and the directive **fdb** 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
endif
```

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
inx                         ; increment X register
endif
```

Note

The **else** and **elsec** directives are equivalent and may 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
inx                      ; increment X register
endc
```

Note

The **elsec** and **else** directives are equivalent and may used without distinction. They are provided for compatibility with previous assemblers.

See Also

if, endc, clist, else

end

Description

Stop the assembly

Syntax

```
end
```

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
inx                       ; increment X register
endc
```

Note

The **endc** and **endif** directives are equivalent and may 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
inx                     ; increment X register
endif
```

Note

The **endif** and **endc** directives are equivalent and may 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
        <macro_body>
        endr
```

Function

The **endr** directive is used to terminate **repeat** sections.

Example

```
        ; shift a value n times
asln:   macro
        repeat  \1
        aslb
        endr
        endm

        ; use of above macro
        asln    10              ;shift 10 times
```

See Also

repeat, repeatl

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.

Example

```
false: equ 0            ; initialize these values
true:  equ 1
tablen:equ tabfin - tabsta;compute table length
nul:   equ $0           ; define strings for ascii charac-
ters
soh:   equ $1
stx:   equ $2
etx:   equ $3
eot:   equ $4
enq:   equ $5
```

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

```
if <expression>     or      if <expression>
instructions                instructions
endif                       else
                            instructions
                            endif
```

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 in the listing depending on the **clist** directive status.

Example

```
if      offset != 1       ; if offset too large
addptr  offset            ; call a macro
else                      ; otherwise
inx                       ; increment X register
endif
```

See Also

else, endif, clist

ifc

Description

Conditional assembly

Syntax

```
ifc <string1>,<string2>  orifc <string1>,<string2>
instructions                 instructions
endc                         elsec
                             instructions
                             endc
```

Function

The **ifc**, **else** 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 in the listing depending on the **clist** directive status.

Example

```
ifc     "hello", \2       ; if "hello" equals argument
ldab    #45               ; load 45
elsec                     ; otherwise...
ldab    #0
endc
```

See Also

elsec, endc, clist

ifdef

Description

Conditional assembly

Syntax

```
ifdef <label>        or        ifdef <label>
instructions                   instructions
endc                           elsec
                               instructions
                               endc
```

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

```
ifeq <expression>     or     ifeq <expression>
instructions                 instructions
endc                         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 in the listing depending on the **clist** directive status.

Example

```
        ifeq    offset          ; if offset nul
        tsta                    ; just test it
        elsec                   ; otherwise
        add     #offset         ; add to accu
        endc
```

See Also

elsec, endc, clist

ifge

Description

Conditional assembly

Syntax

```
ifge <expression>     or     ifge <expression>
instructions                 instructions
endc                         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 in the listing depending on the **clist** directive status.

Example

```
ifge    offset-127      ; if offset too large
addptr  offset          ; call a macro
elsec                   ; otherwise
inx                     ; increment X register
endc
```

See Also

elsec, *endc*, *clist*

ifgt

Description

Conditional assembly

Syntax

```
ifgt <expression>     or     ifgt <expression>
instructions                 instructions
endc                         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 in the listing depending on the **clist** directive status.

Example

```
ifgt    offset-127        ; if offset too large
addptr  offset            ; call a macro
elsec                     ; otherwise
inx                       ; increment X register
endc
```

See Also

elsec, *endc*, *clist*

ifle

Description

Conditional assembly

Syntax

```
ifle <expression>     or     ifle <expression>
instructions                 instructions
endc                         elsec
                             instructions
                             endc
```

Function

The **ifle**, **elsec** and **endc** directives allow conditional assembly. The **ifle** 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 **ifle** 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 **ifle** 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 in the listing depending on the **clist** directive status.

Example

```
ifle    offset-127      ; if offset small enough
inx                     ; increment X register
elsec                   ; otherwise
addptr  offset          ; call a macro
endc
```

See Also

elsec, endc, clist

iflt

Description

Conditional assembly

Syntax

```
iflt <expression>    or    iflt <expression>
instructions               instructions
endc                       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 in the listing depending on the **clist** directive status.

Example

```
iflt     offset-127        ; if offset small enough
inx                        ; increment X register
elsec                      ; otherwise
addptr   offset            ; call a macro
endc
```

See Also

elsec, *endc*, *clist*

ifnc

Description

Conditional assembly

Syntax

```
ifnc <string1>,string2> or ifnc <string1><string2>
instructions                instructions
endc                        elsec
                            instructions
                            endc
```

Function

The **ifnc**, **elsec** and **endc** directives allow conditional assembly. The **ifnc** directive is followed by a constant expression. If *<string1>* and *<string2>* are differents, 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 in the listing depending on the **clist** directive status.

Example

```
ifnc    "hello", \2
addptr  offset          ; call a macro
else                    ; otherwise
inx                     ; increment X register
endif
```

See Also

elsec, endc, clist

ifndef

Description

Conditional assembly

Syntax

```
ifndef <label>   or    ifndef <label>
instructions           instructions
endc                   elsec
                       instructions
                       endc
```

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

```
ifne <expression>      or    ifne <expression>
instructions                 instructions
endc                         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 in the listing depending on the **clist** directive status.

Example

```
ifne    offset          ; if offset not nul
add     #offset         ; add to accu
elsec                   ; otherwise
tsta                    ; just test it
endc
```

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

```
list
```

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”
        ldx     nbr     ; expand as ‘ldx #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$:    ldab    var
        beq     10$
        stab    var2
local
10$:    ldaa    var2
        beq     10$
        staa    var
        rts
```


macro

Description

Define a macro

Syntax

```
label: macro    <argument_list>
        <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 227.

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"
        ldaa     _#2
        staa     _SCR
```

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
        jsr     \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 <expresion>
```

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 is 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:
       ldy     next,x           ; 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

```
page
```

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 &lt;expression&gt;
       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.

A **repeat** directive may be terminated early by using the **rexit** directive which, when encountered, acts as if the end of the **repeat** has been reached.

Example

```
; shift a value n times
asln:   macro
        repeat  \1
        aslb
        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
        addd    #\1     ; add to accu
endr
end
```

or

```
; test a value using the named syntax
repeatl \count,1,2,3
        addd    #\count ; add to accu
endr
end
```

will both produce:

```
 2              ; test a value
 9  0000 c30001addd     #1       ; add to accu
 9  0003 c30002addd     #2       ; add to accu
 9  0006 c30003addd     #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$:    ldab     var
        beq      10$
        stab     var2
function2:
10$:    ldaa     var2
        suba     var
        bne      10$
        rts
        restore
var3:   ds.b     1
var4:   ds.b     1

        switch   .text

        ldaa     var3
        staa     var4
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
         aslb
         endr
         endm

; use of above macro
         asln     5
```

See Also

endr, repeat, repeatl

save

Description

Save section

Syntax

```
save
```

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$:    ldab    var
        beq     10$
        stab    var2
function2:
10$:    ldaa    var2
        suba    var
        bne     10$
        rts
        restore
var3:   ds.b    1
var4:   ds.b    1

        switch  .text

        ldaa    var3
        staa    var4
end
```

See Also

restore, section

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:

abs	absolute section
bss	bss style section (no data)
hilo	values are stored in descending order of significance
even	enforce even starting address and size
zpage	enforce 8 bit relocation
long	enforce 32 bit relocation

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 to set the default page header used when a new page is written to the listing output.

Example

```
        title    "My Application"
```

See Also

messg, page, plen

Note

For compatibility with previous assemblers, the directive **ttl** is alias to *title*.

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

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

xdef

xref.5

Description

Declare a special external symbol

Syntax

```
xref.5 identifier[,identifier...]
```

Function

The directive **xref.5** declares external symbols to be handled as **5** bits signed values, allowing the assembler to encode an indexed addressing mode with the smallest size as possible. The linker will verify that the final value is compatible with the encoded addressing mode, and will output an error message if not.

Example

```
xref.5   small

ldd      small,x  ; short offset
```

See Also

xref, xref.9

xref.9

Description

Declare a special external symbol

Syntax

```
xref.9 identifier[,identifier...]
```

Function

The directive **xref.9** declares external symbols to be handled as **9** bits signed values, allowing the assembler to encode an indexed addressing mode with the appropriate size. The linker will verify that the final value is compatible with the encoded addressing mode, and will output an error message if not.

Example

```
xref.9  medium

ldd     medium,x ; one byte offset
```

See Also

xref, xref.9

wdef

Description

Declare a symbol as a default definition

Syntax

```
wdef identifier[,identifier...]
```

Function

The **wdef** directive allows declaring a symbol as public with a weak definition, meaning that another definition of the same symbol in an other file using an *xdef* directive, will silently overwrite the *wdef* definition.

A symbol may be declared both *wdef* and *xref* in the same module, to allow for usage of common headers.

Example

```
        wdef     it_irq   ; default irq handler
it_irq:
        b        it_irq   ; stay here
```

See Also

xdef

CHAPTER

6

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:

6

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 307.

- 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 309.

- 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
(**+seg** *<section>*) controls the creation of new segments and has parameters which are selected from the set of local flags. (**+grp** *<section>*) controls the section grouping.
Group 2
(**+inc***) is used to include files
Group 3
(**+new**, **+pub** and **+pri**) controls name regions and takes no parameters.
Group 4
(**+def** *<symbol>*) is used to define symbols and aliases and takes one required parameter, a string of the form **ident1=ident2**, a string of the form **ident1=constant**, or a string of the form **ident1=@segment**.
Group 5
(**+spc** *<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 .data -b0x2000             # start data address
+seg .text -b0xe000 -n .text    # start eprom address
+seg .const -a .text            # constants follow program
\cx12\lib\crts.h12              # startup object file
mod1.o mod2.o                   # input object files
\cx12\lib\libi.h12              # C library
\cx12\lib\libm.h12              # machine library
+seg .const -b0xffce            # vectors eprom 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
-bs#	set the window shift to #, which implies that the number of bytes in a window is **2**#**. The default value is **14** (bank switching enabled). For more information, see the section "*Address Specification*" on page 318.
-e*	log errors in the text file * instead of displaying the messages on the terminal screen.
-ge	produce error messages directly compatible with the Eclipse environment
-l*>	specify library path. You can specify up to 128 different paths. Each path is a directory name, **not** terminated by any directory separator character.
-m*	produce map information for the program being built to file *.
-o*	write output to the file *. This option is required and has no default value.
-p	display symbols with physical address instead of logical address in the map file.
-s	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.
-sa	display symbols sort by address instead of alphabetic order in the map file.
-si	suppress the **.info.** segment content for compatibility with tools not supporting this segment yet.
-sl	output local symbols in the executable file.

Global linker Options

Option	Description
-u#	display unused symbols. Valid options are: **-u1** – display data symbols **-u2** – display code and constant symbols **-u4** – display absolute symbols (located variables) **-u8** – 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.
-v	be "verbose".

!! IMPORTANT !!

*Applications **not** using bank switching should specify the **-bs0** option to disable the internal banking verification.*

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
-a*	make the current segment follow the segment *****, where ***** refers to a segment name given explicitly by a **-n** option. Options **-b** and **-e** cannot be specified if **-a** has been specified. Option **-o** can be specified only with value **0** to reset the logical address to the same value than the physical address.
-b*	set the physical start address of the segment to *. Option **-e** or **-a** cannot be specified if **-b** has been specified.
-c	do not output any code/data for the segment.
-ck	mark the segment you want to check. For more information, see "*Checksum Computation*" on page 331.
-cs*	define a count space name for the segment. The total amount of bytes counted in all segments sharing the same space name is displayed in the map file.
-ds#	set the bank size for paged addresses calculation. This option overwrites the global **-bs** option for that segment.
-e*	set the physical end address of the segment to *. Option **-b** or **-a** cannot be specified if **-e** has been specified.

Segment Control Options Usage (cont.)

Option	Description
-f#	fill the segment up to the value specified by the **-m** option with single bytes or two byte words whose value is #. This option has no effect if no **-m** option is specified for that segment.
-i?	define the initialization option. Valid options are: **-it** — use this segment to host the descriptor and images copies of initialized data used for automatic data initialization **-id** — initialize this segment **-ib** — do not initialize this segment **-ik** — mark this segment as checksum segment **-ic** — mark this segment as moveable segment
-k	mark the segment as a root segment for the unused section suppression. This flags 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.
-m*	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 **-a** option as following a segment which is marked with the **-m** option, then set the maximum available space for all the possible consecutive segments. If a **-m** is specified on a **-a** 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 **-a** 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

-n*	set the output name of the segment to *. Segment output names have at most **15** characters; longer names are truncated. If no name is given with a **-n** option, the segment inheritates a default name equal to its assembler section name.
	For example, use this option when you want to generate the hex records for a particular PROM, such as:
	`+seg .text -b0x2000 -n prom1`
	`<object_files>`
	`+seg .text -b0x4000 -n prom2`
	`<object_files>`
	`...`
	You can generate the hex records for **`prom1`** by typing:
	`chex -n prom1 file.h12`
	For more information, see "*The chex Utility*" in **Chapter 8**.

-o*	set the logical start address of the segment to * if **-b** option is specified or the logical end address if **-e** option is specified. The default is to set the logical address equal to the physical address. Options **-b** and **-e** cannot be specified both if **-o** has been specified.
-q*	enable a manual initialization mechanism by specifying a name of an empty segment which will host the initialization image of the current segment. See "*Manual Segment Initialization*" on page 330.
-r*	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 **-r3** 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** option and has no effect on the first object module.

Segment Control Options Usage (cont.)

Option	Description
-s*	define a space name for the segment. This segment will be verified for overlapping only against segments defined with the same space name. See "*Overlapping Control*" on page 318.
-v	do not verify overlapping for the segment.
-w*	set the window size for banked applications, and activate the automatic bank segment creation.
-x	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 **-m** option. The next segment must be declared **before** the object causing the overflow. This option has no effect if no **-m** option is specified for the expandable segment. Option **-e** cannot be specified with option **-x**.

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 316.

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.

@1, @2,...	include each individual object file at its positional location on the command line and insert them at the respective locations in the link file (**@1** is the first object file, and so on).
@*	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.h12 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.h12 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:

+inc*	include the file specified by *****. This is equivalent to expanding the text file into the link file directly at the location of the **+inc** 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:

+new	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.
+pub	make the following portion of a given region public.
+pri	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:

ident=constant	where *ident* is a valid identifier and *constant* is a valid constant expressed with the standard C language syntax. This form is used to add *ident* to the symbol table as a defined absolute symbol with a value equal to *constant*.
ident1=ident2	where *ident1* and *ident2* are both valid identifiers. This form is used to define aliases. The symbol *ident1* is defined as the alias for the symbol *ident2* and goes in the symbol table as an external DEF (a DEF is an entity defined by a given module.) If *ident2* 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).
ident=@section	where *ident* is a valid identifier, and *section* is the name of a section specified as the first argument of a **+seg** directive. This form is used to add *ident* 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.

ident=start(segment)	This form defines *ident* as the *logical start* address of the designated segment.
ident=end(segment)	This form defines *ident* as the *logical end* address of the designated segment.
ident=pstart(segment)	This form defines *ident* as the *physical start* address of the designated segment.

ident=pend(segment)	This form defines *ident* as the *physical end* address of the designated segment.
ident=size(segment)	This form defines *ident* as the *size* of the designated segment.
ident=init(segment)	This form defines *ident* as the *initialization image start* 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 333.

Reserve Space Option

The following option is used to reserve space in a given segment:

+spc <segment>=<value>	reserve <*value*> bytes of space at the current location in the segment named <*segment*>.
+spc <segment>=@section	reserve a space at the current location in the segment named <*segment*> equal to the current size of the opened segment where the given *section* 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 '='.*

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.

In the paged architecture of the HC12/HCS12, the *bias* is the physical address of the start of the segment in question, as seen from memory. The *offset* is the logical address of the start of the segment, as seen from the processor.

The window shift specified by the **-bs#** option gives a measure of the resolution used to hold the *bias* value of a segment. If the value specified by the **-bs#** option is **n**, then the resolution is **2******n**. For example, the value of **n** is **14** for the **HC12/S12**.

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*,and *.bsct*). By default, *.text* and *.bsct* segments start at zero, *.ubsct* segment follows the latest *.bsct* 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 *.const.w* when **+ceven** is selected).

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 place ***after*** all objects that may call it's modules to avoid unresolved references. The standard ANSI libraries are provided in three 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 double precision floating point support is not needed. The first letter after "*lib*" in each library file denotes the library type (**d** for double, **f** for single precision, and **i** for integer). See below.

libd.h12	Double Precision Library provides ANSI double precision floating point support. Link this library ***before*** the other libraries when needed.

libf.h12	Single Precision Library. Used in conjunction with the **+sprec** option to force all floats (even variables declared as doubles) to single precision. This library is used for applications where only single precision floating point support is needed. This library is significantly smaller and faster than the double precision. Link this library ***before*** the other libraries when ***only*** single precision floats are used. **NOTE** *The **+sprec** compiler option **MUST** be used if you want to use the Single Precision library in order to suppress normal ANSI float to double promotions.*
libi.h12	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.
libb.h12	Basic Eeprom Library. This library is designed to give access to eeprom functions of the first HC12 processors. When used, this library **MUST** be linked ***before*** any other libraries. **NOTE** *This library is only necessary for existing applications built for non Star12 processor with previous versions of the compiler.*
libe.h12	Standard Eeprom Library. This library is designed to give access to the HCS12 eeprom functions. When used, this library **MUST** be linked ***before*** any other libraries.
fuzzy.h12	Fuzzy Library. This library is designed to give access to the specific fuzzy instructions of the HC12/HCS12. This library does not depend on the others and maybe located regardless of the other ones position.

Machine Library	DP256 eeprom Library	Integer Only Library	Single Precision Floats	Double Precision Floats
Libm.h12	Libe.h12	libi.h12	libf.h12	libd.h12

NOTE

Compiler libraries must be located in a non-banked area of memory or duplicated in each bank that uses them.

Machine Library	DP256 eeprom Library	Integer Only Library	Single Precision Floats	Double Precision Floats
Libm.h12	Libe.h12	libi.h12	libf.h12	libd.h12

NOTE

Compiler libraries must be located in a non-banked area of memory or duplicated in each bank that uses them.

Library Order

You should link your application with the libraries in the following orders:

Integer Only Application	Single Precision Float Application	Double Precision Float Application
(libb.h12)	(libb.h12)	(libb.h12)
libi.h12	libf.h12	libd.h12
libm.h12	libi.h12	libi.h12
	libm.h12	libm.h12

NOTE

*The **libb.h12** is only necessary when building applications using eeprom functions and targeting the HC12 derivative (none Star12 processor).*

For more information, see “*Linker Command Line Examples*” on page 342.

Alternate Machine Library

The standard machine library (**libm.h12**) implements the basic floating point operations without checking underflow and overflow for efficiency reasons. If the application needs standard behaviour regarding underflow and overflow, the machine library must be replaced by **libmc.h12**.

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
```


Bank Switching

The linker is able to build banked segments for large applications. Such banks can be built explicitly, or automatically. A banked segment is described by a physical start address, specified by the **-b** option, a logical start address, specified by the **-o** option, and a window size. The logical address is the processor address, and should match the windowed area (**`0x8000`** to **`0xbfff`** for the HC12/HCS12). The physical address is the memory address and should match the hardware specifications.

A single bank is defined by using the **-b** and **-o** only. The bank size should be specified by the **-m** option to check any bank overflow. Several banks can be defined by several independent segment directives.

Multiple banks are automatically defined by using the **-b**, **-o** and **-w** options. The bank size is defined by the **-w** option which also sets up the automatic filling mechanism. The **-m** option still can be used, but then defines the maximum available space for all the possible consecutive banks. When automatic filling is activated, a new segment is started when the current bank size exceeds the value given by the **-w** option. The new bank physical start address is obtained by adding the window size to the current bank physical start address. The new logical start address is equal to the current bank logical start address. If a maximum size has been specified for the current bank by the **-m** option, a maximum size is defined for the new bank with a new value obtained by substracting the window size to the current bank maximum size.

Here is an example for a link file using single banks:

```
# Link for EPROM
+seg .data -b0x2000      # start data address
+seg .ftext -b 0x10000 -o 0x8000 -m 0x4000
func1.o func2.o func3.o
+seg .ftext -b 0x14000 -o 0x8000 -m 0x4000
func4.o func5.o func6.o
+seg .text -b0xc000 -o0xc000 -n.text# start eprom address
+seg .const -a .text     # constants follow code
\cx\lib\crts.h12         # startup object file
mod1.o mod2.o            # input object files
\cx\lib\libi.h12         # C library
\cx\lib\libm.h12         # machine library
+seg .const -b0xffce     # vectors eprom address
vectors.o                # reset and interrupt vectors
```

The following link file shows the use of a multiple banks: the **-w** option specifies a size of the window (**0x4000**). In this case, when the current bank size exceeds, a new segment is created with a logical start address of **0x8000**, and the new physical address will be **0xc000**.

```
# Link for EPROM
+seg .data -b0x2000     # start data address
+seg .ftext -b 0x10000 -o 0x8000 -m 0x8000 -w0x4000
func1.o func2.o func3.o
func4.o func5.o func6.o
+seg .text -b0xc000 -o0xc000 -n.text# start eprom address
+seg .const -a .text    # constants follow code
\cx\lib\crts.h12        # startup object file
mod1.o mod2.o           # input object files
\cx\lib\libi.h12        # C library
\cx\lib\libm.h12        # machine library
+seg .const -b0xffce    # vectors eprom address
vectors.o               # reset and interrupt vectors
```

The linker also verifies that a bank is properly entered with a **call** instruction. Any attempt to enter a bank with a **jsr** instruction will be reported as an error, unless the **jsr** is issued from the same bank.

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 *.bsct* 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 paged_ram_address    ; paged value of ram start address
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 *crtsi.s* startup routine. So all this mechanism will be activated just by linking the *crtsi.h12* 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 *crtsi.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 *crtsi.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:

_checksum() **_checksumf()**	check a 8 bit checksum stored once for all the selected segments.
_checksumx() **_checksumxf()**	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.
_checksum16() **_checksum16f()**	check a 16 bit checksum stored once for all the selected segments.
_checksum16x() **_checksum16xf()**	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.
_checkcrc16()	check a 16 bit checksum stored once for all the selected segments using CCITT algorithm.
_checkcrc16x()	check a 16 bit checksum stored for every selected segments using CCITT algorithm. This method allows a segment to be dynamically reloaded by updating the corresponding CRC word.

Each function is provided in a compact version and a fast version (name ending with '**f**'). The compact version consumes less code and the fast version executes faster.

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
crtsi.h12
test.o
libi.h12
libm.h12
```

The descriptor built by the linker is a list of entries followed by the expected CRC value, only once if functions *_checksum()*, or *_checksum16() _checkcrc16()* are called, or after each entry if functions *_checksumx()* or *_checksum16x() or checkcrc16x()* 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() or checkcrc16()* 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 _checksum type algorithm is the following. 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: **REF**s and **DEF**s. 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 an HC12/HCS12 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.h12 -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 the *Segment* section, the *Count* section (optional), the *Modules* section, the *Stack Usage* section, the *Call Tree* 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.

Counts List the total amount of bytes produced in each count space defined in the linker command file. If no count space is defined, this section is not displayed.

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.

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 HC12/HCS12 file from *file.o* you can type at the system prompt:

```
clnk -o myapp.h12 myapp.lkf
```

where *myapp.lkf* contains:

```
+seg .text -b 0xf000 -n.text  # program start address
+seg .const -a .text          # constants follow program
+seg .data -b0x800            # start data address
+def __sbss=@.bss             # symbol used by startup
\cx12\lib\crts.h12            # startup object file
file1.o file2.o               # input object files
\cx12\lib\libi.h12            # C library
\cx12\lib\libm.h12            # 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 .data -b0x800            # start data address
+def __sbss=@.bss             # symbol used by startup
\cx12\lib\crts.h12            # startup with NO-INIT
acia.o                        # main program
module1.o                     # module program
\cx12\lib\libi.h12            # integer library
\cx12\lib\libm.h12            # machine library
+seg .const -b0xffce          # vectors eprom address
vector.o                      # reset & interrupt vectors
+def __memory=@.bss           # symbol used by library
+def __stack=0x1000           # stack pointer initial value
```


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 -b 0xf000 -n.text  # program start address
+seg .const -a .text          # constants follow program
+seg .data -b0x800            # start data address
+def __sbss=@.bss             # symbol used by startup
\cx12\lib\crtsi.h12           # startup with auto-init
acia.o                        # main program
module1.o                     # module program
\cx12\lib\libf.h12            # single prec.
\cx12\lib\libi.h12            # integer library
\cx12\lib\libm.h12            # machine library
+seg .const -b0xffce          # vectors eprom address
vector.o                      # reset & interrupt vectors
+def __memory=@.bss           # end of bss segment
+def __stack=0x1000           # stack pointer initial value
```

CHAPTER

7

Debugging Support

This chapter describes the debugging support available with the cross compiler targeting the HC12/HCS12. 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

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
-fc*	print debugging information only about the function *. By default, *cprd* prints debugging information on all functions in *<file>*. Note that information about global data objects is always displayed when available.
-fl*	print debugging information only about the file *. By default, *cprd* prints debugging information on all C source files.
-o*	print debugging information to file *. Debugging information is written to your terminal screen by default.

Cprd Option Usage (cont.)

Option	Description
-ra	display any occurence of structure fields with their offset.
-r	display the first occurence of structure fields with their offset.
-s	display object size in bytes.
-u	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.h12
```

```
Information extracted from acia.h12

source file acia.c:

unsigned char buffer[512] at 0x0804
unsigned char *ptlec at 0x0802
unsigned char *ptecr at 0x0800

unsigned char getch() lines 26 to 36 at 0xf016-0xf02b
    auto unsigned char c at -1 from frame pointer

void outch() lines 40 to 45 at 0xf02c-0xf032
    argument unsigned char c at 1 from frame pointer

void recept() lines 51 to 57 at 0xf033-0xf047
    (no locals)

void main() lines 63 to 71 at 0xf048-0xf061
    (no locals)

source file vectors.c:

void (*_vectab[25])() at 0x3ffce
```

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
-a	when set, cause *clst* to list files in alphabetical order. The default is that they are listed by increasing addresses.
-b	display physical address instead of logical address in the listing file.
-f*>	specify * 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
-i*>	read string ***** 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 *clst*. You can specify up to 10 different paths Each path is a directory name, **not** terminated by any directory separator character.
-l#	when paginating output, make the listings # lines long. By default, listings are paginated at 66 lines per page.

-o*	redirect output from *clst* to file *****. You can achieve a similar effect by redirecting output in the command line.
	`clst -o acia.lst acia.h12`
	is equivalent to:
	`clst acia.h12>acia.lst`

-p	suppress pagination. No page breaks will be output.
-r#:#	where **#:#** is a range specification. It must be of the form *<number>:<number>*. 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 **-r**: `-r 0xe000:0xe200`

CHAPTER

8

Programming Support

This chapter describes each of the programming support utilities packaged with the C cross compiler targeting the HC12/HCS12. The following utilities are available:

Utility	Description
cbank	fill page window
chex	translate object module format
clabs	generate absolute listings
clib	build and maintains libraries
cobj	examine objects modules
cv695	generate IEEE695 format
cvdwarf	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 cbank Utility

You use the **cbank** utility to optimize the bank filling with object files. *cbank* is given a list of object files and a bank size. It reorganizes the object list in order to fill as completely as possible the smallest amount of banks and produces as result a text file containing the object file names in the proper order. If the input file also contains bank start addresses (using the linker syntax), segment opening directives will be also output at the proper place with the specified information. Otherwise the object file list is supposed to be used in conjunction with the **-w** option of the linker allowing an automatic bank filling. In any cases, the file produced by the cbank utility can be directly inserted in the linker command file by a **+inc** directive.

Command Line Options

cbank accepts the following command line options, each of which is described in detail below:

```
cbank [options] file
      -d     discard unused files
      -m#    maximum available banks
      -n*    name of segment to pack
      -o*    output file name
      -w##   bank size
```

Cbank Option Usage

Option	Description
-d	discard unused files. By default, *cbank* outputs all input files, those not used beeing displayed at the end of the list.
-m#	fill a maximum of # banks. If cbank needs more banks than the specified number, it will report an error message. By default, *cbank* fills as many banks as necessary.
-n*	sort sections whose name is equal to the string *. By default, *cbank* sorts **.text** sections.
-o*	write result to file *. The default is STDOUT.
-w##	set the bank size to ##.

Return Status

cbank returns success if no error messages are printed. Otherwise it returns failure.

Examples

The following command:

```
cbank -o bk_list -w 0x1000 obj_list
```

will generate `bk_list` as the result file, with a page window of size `0x1000` from the given list `obj_list` which contains:

```
file1.o
file2.o
file3.o
file4.o
```

The result will be:

```
# --- bank 1 --- # (3876/4096)
file1.o
file3.o
# --- bank 2 --- # (3900/4096)
file2.o
# --- bank 3 --- # (474/4096)
file4.o
```

The first value is the space used in the bank, and the second value is the bank size.

Bank start addresses can be included into the input file, such as:

```
-b0x10000 -o 0x8000 -n bank1
-b0x18000 -o 0x8000 -n bank2
-b0x20000 -o 0x8000 -n bank3
file1.o
file2.o
file3.o
file4.o
```

The result will be:

```
+seg .text -b0x10000 -o0x8000 -n bank1 # (3876/4096)
file1.o
file3.o
+seg .text -b0x18000 -o0x8000 -n bank2 # (3900/4096)
```



```
file2.o
+seg .text -b0x20000 -o0x8000 -n bank3 # (474/4096)
file4.o
```

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
-a##	the argument file is a considered as a pure binary file and ## is the output address of the first byte.
-b##	substract ## to any address before output.

Chex Option Usage (cont.)

Option	Description
-e##	define ## as the entry point address encoded in the dedicated record of the output format, if available.
-f?	define output file format. Valid options are: **i** Intel Hex Format **m** Motorola S19 format **2** Motorola S2 format **3** Motorola S3 format Default is to produced Motorola S-Records (**-fm**). Any other letter will select the default format
-h	do not output the header sequence if such a sequence exists for the selected format.
+h*	insert * in the header sequence if such a sequence exists for the selected format.
-m#	output # maximum data bytes per line. Default is to output 32 bytes per line.
-n*>	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 **-n** option of the linker.
-o*	write output module to file *. The default is STDOUT.
-p	output addresses of banked segments using a paged format `<page_number><logical_address>`, instead of the default format `<physical>`.
-pa	output addresses of banked data segments using a paged format `<page_number><logical_address>`, instead of the default format `<physical>`.

Chex Option Usage (cont.)

Option	Description
-pl##	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.
-pn	behaves as **-p** 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.
-pp	behaves as **-p** 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.
-s	sort the output addresses in increasing order.
-w	output word addresses. Addresses must be aligned on even addresses. This option is useful for word processor type.
-x*>	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
```

```
:0E000000020068656C6C6F20776F726C640094
: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
-a	process also files located in libraries. Default is to process only all the files of the application.
-cl*	specify a path for the listing files. By default, listings are created in the same directory than the source files.
-l	process files in the current directory only. Default is to process all the files of the application.

Clabs Option Usage (cont.)

Option	Description
-p	output addresses of banked segments using a paged format `<page_number><logical_address>`, instead of the default format `<physical>`.
-pn	behaves as **-p** but only when logical address is inside the banked area.
-pp	behaves as **-p** but uses paged addresses for all banked segments, mapped or unmapped.
-r*	specify the input suffix, including or not the dot '.' character. Default is "**.ls**"
-s*	specify the output suffix, including or not the dot '.' character. Default is "**.la**"
-v	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.h12
```

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.h12
```

will look for files with the suffix “**.lst**”:

The following command line:

```
clabs -s.lx acia.h12
```

will generate:

```
crts.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
-a	include absolute symbols in the library symbol table.
-c	create a library containing <*files*>. Any existing <*library*> of the same name is removed before the new one is created.
-d	delete from the library the zero or more files in <*files*>.
-e	accept module with no symbol.
-i*	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 <*files*>, then <*files*> will be also added to the library.

Clib Option Usage (cont.)

Option	Description
-l	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.
-p	do not prefix module names in library with any path. This option is only meaningfull with option **-c**.
-r	in an existing library, replace the zero or more files in <*files*>. If no library <*library*> exists, create a library containing <*files*>. The files in <*files*> not present in the library are added to it.
-s	list the symbols defined in the library with the module name to which they belong.
-t	list the files in the library.
-v	be verbose
-x	extract the files in <*files*> that are present in the library into discrete files with the same names. If no <*files*> 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.

```
cobj [options] file
     -d     output data flows
     -h     output header
     -i     display info section
     -n     output sections
     -o*    output file name
     -r     output relocation flows
     -s     output symbol table
     -v     display file addresses
     -x     output debug symbols
```

<file> specifies a file, which must be in relocatable format or executable format.

Cobj Option Usage

Option	Description
-d	output in hexadecimal the data part of each section.
-h	display all the fields of the object file header.
-i	display the content of the **.info.** section in a readable format.
-n	display the name, size and attribute of each section.
-o*	write output module to file *****. The default is STDOUT.
-r	output in symbolic form the relocation part of each section.
-s	display the symbol table.
-v	display seek addresses inside the object file.
-x	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 cv695 Utility

cv695 is the utility used to convert a file produced by the linker into an IEEE695 format file.

Command Line Options

cv695 accepts the following options, each of which is described in detail below.

```
cv695 [options] file
        +V4     do not offset locals
        +bit    patch bit variables into chars
        -d      display usage info
        +dpage  file uses data paging (HC12 only)
        -mod?   select compiler model
        +old    produce old format
        -o*     output file name
        +page#  define pagination (HC12 only)
        -rb     reverse bitfield (L to R)
        -v      be verbose
```

<*file*> specifies a file, which must be in executable format.

Cv695 Option Usage

Option	Description
-V4	output information as per as *cv695* converter V4.x version. This flag is provided for compatibility with older version of *cv695* version. **DO NOT USE UNLESS SPECIFICALLY INSTRUCTION TO DO SO**.
+bit	patch bit variables into chars because IEEE695 format does not handle bit variables.
+dpage	output banked data addresses. **DO NOT USE THIS OPTION ON NON BANKED DATA APPLICATION. THIS FLAG IS CURRENTLY ONLY MEANINGFULL FOR THE HC12/HCS12.**

Cv695 Option Usage (cont.)

Option	Description
-d	dump to the screen the interface information such as: frame coding, register coding, *e.g.* all the processor specific coding for IEEE (note: some of these codings have been chosen by COSMIC because no specifications exist for them in the current published standard). **THIS INFORMATION IS ONLY RELEVANT FOR WRITING A READER OF THE PRODUCED IEEE FORMAT**.
-mod?	where **?** is a character used to specify the compilation model selected for the file to be converted. **THIS FLAG IS CURRENTLY ONLY MEANINGFULL FOR THE HC16.** This flag mimics the flag used with C. Acceptable values are: **c** for compact model **s** for short model **t** for tiny model **l** for large model
+old	output old format for MRI.
-o*	where ***** is a filename. ***** is used to specify the output file for *cv695*. By default, if **-o** is not specified, *cv695* send its output to the file whose name is obtained from the input file by replacing the filename extension with “.695”.

Cv695 Option Usage (cont.)

Option	Description
+page#	output addresses in paged mode where # specifies the page type: <table><tr><td>**0**</td><td>for no paging</td></tr><tr><td>**1**</td><td>for pages with PHYSICAL ADDRESSES</td></tr><tr><td>**2**</td><td>for pages with banked addresses <page><offset_in_page></td></tr></table> By default linear physical addresses are output. **THIS FLAG IS CURRENTLY ONLY MEANINGFULL FOR THE HC12/HCS12.**
-rb	reverse bitfield from left to right.
-v	select verbose mode. *cv695* will display information about its activity.

Return Status

cv695 returns success if no problems are encountered. Otherwise it returns failure.

Examples

Under MS/DOS, the command could be:

```
cv695 C:\test\acia.h12
```

and will produce: `C:\test\acia.695`

and the following command:

```
cv695 -o file C:\test\acia.h12
```

will produce: `file`

Under UNIX, the command could be:

```
cv695 /test/acia.h12
```

and will produce: **test/acia.695**

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
-bp#	start address of the banking page.
-bs#	set the window shift to #, which implies that the number of bytes in a window is **2**#**. **THESE FLAGS ARE CURRENTLY ONLY MEANINGFULL FOR THE HC11K4.**
+dup	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.

Cvdwarf Option usage (cont.)

Option	Description
-e*	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 **__stext** is used by default.
-loc	location lists are used in place of location expressions whenever the object whose location is being described can change location during its lifetime. **THIS POSSIBILITY IS NOT SUPPORTED BY ALL DEBUGGERS.**
-o*	where ***** is a filename. ***** is used to specify the output file for *cvdwarf*. By default, if **-o** is not specified, *cvdwarf* send its output to the file whose name is obtained from the input file by replacing the filename extension with "**.elf**".

Cvdwarf Option usage (cont.)

Option	Description
+page#	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> 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.**
-rb	reverse bitfield from left to right.
-so	add stack offset. This option has to be selected when using debuggers using the SP value directly. **THIS FLAG IS CURRENTLY ONLY MEANINGFULL FOR THE HC08/HCS08.**
-v	select verbose mode. *cvdwarf* 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:

```
cvdwarf C:\test\acia.h12
```

and will produce: **`C:\test\acia.elf`**

and the following command:

```
cvdwarf -o file C:\test\acia.h12
```

will produce: **`file`**

Under UNIX, the command could be:

```
cvdwarf /test/acia.h12
```

and will produce: **`test/acia.elf`**

APPENDIX

A

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 (cp6812) 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 beeing 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 assignment 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 in 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 substracted

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 substraction

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 simple 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**)

<name> used before set - a local variable has been used before beeing initialized by any previous code (**+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 (cg6812) 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@nostack 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 *cp6812* you used to compile your program is incompatible with *cg6812*.

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 (ca6812) 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.

bank crossing call - a *jsr* instruction has been used to enter a banked function, either from a different bank or from a common area.

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

section <segment> larger than bank size - the size of a segment is larger than the maximum value allowed by the **–bs** option

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.

unexpected bank location - an interrupt function or a function accessing the PPAGE register is located in a bank.

unknown directive - a directive name has not been recognized as a linker directive.

APPENDIX

B

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 **cx6812.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 **cx6812**.

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 **cx6812.cxf** as provided by the default installation appears below:

```
# CONFIGURATION FILE FOR HC12/HCS12 COMPILER
# Copyright (c) 1996 by COSMIC Software
#
-pu                       # unsigned char
-i c:\cosmic\h6812        # include path
-ans                      # default assembler to HCS12
-m debug:x                # debug: produce debug info
-m even:b                 # even:align data on even boundary
-m ceven:,cs              # ceven: use two const sections
-m fast:,i                # fast: inline long machine calls
-m modf:hmodf.h,,,,modff,fast # all functions are far

-m nobss:,bss             # nobss: do not use bss
-m nocst:,ct              # nocst: constant in text section
-m nofds:,df              # nofds: do not use far data section
-m nofts:,tf              # nofts: do not use far text section
-m "nowiden:nw -p"        # nowiden: do not expand argument
-m pic:,cr1,,dPIC,picd,picds# pic: position independant code
-m picd:,cr7,,dPIC,pic,picds# picd:pos. indep. code and data
-m picds:,cr3,,dPIC,pic,picd# picds: pic and data separated
-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 sprec:f                # use float only
-m zpage:hzpage.h         # zpage: @dir for all variables
-m pgff:,t1,,na,std       # pgff: enable old dx128 processors
```



```
-m hcs:,,,,pgff,std,mcs # enable hcs12 processor (default)
-m mcs:,,,,pgff,std,hcs # enable hcs12 processors (default)
-m std:,t0,,na,pgff,hcs,mcs# std: enable standard processor
-m warn:w1              # warn: enable warnings
```

The following command line:

```
cx6812 hello.c
```

in combination with the above configuration file directs the **cx6812** compiler to execute the following commands:

```
cp6812 -o \2.cx1  -u  -i\cosmic\h6812 hello.c
cg6812 -o \2.cx2 \2.cx1
co6812 -o \2.cx1 \2.cx2
ca6812 -o hello.o -i\cosmic\h6812 \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**ssembler) 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:l,dl1,,,dl2# dl1: line option 1
-m dl2:l,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

```
cx6812 +nobss +rev hello.c
```

in combination with the previous configuration file directs the **cx6812** compiler to execute the following commands:

```
cp6812 -o \2.cx1  -u  -rb -i\cosmic\h6812 hello.c
cg6812 -o \2.cx2 -bss \2.cx1
co6812 -o \2.cx1 \2.cx2
ca6812 -o hello.o -i\cosmic\h6812 \2.cx1
```

APPENDIX

C

HC12/HCS12 Machine Library

This appendix describes each of the functions in the Machine Library (**libm**). These functions provide the interface between the HC12/HCS12 microcontroller hardware and the functions required by the code generator. They are described in reference form, and listed alphabetically.

Note that machine library functions return values as follows:

- **integer** in **D** register.
- **longs** and **floats** in a register pair (“float register” or “long register” depending on context) whose low word is the **D** register and whose high word is in the **X** register.
- **pointer to long**, **float** or **double** in **X** or **Y** register.
- **far pointer** in a register pair whose low word (offset) is the **X** register and whose high word is the **D** register, the page number in **A** register, **B** register beeing always zero.

In the functions description below, left and right refer to left and right operands, or first and second operands, of library functions.

c_bfget

Description

Get a long bitfield

Syntax

```
; raw value in long register
     jsr   c_bfget
     dc.l  mask
; result in long register
```

Function

c_bfget is extracting a long bitfield from the value loaded in the long register using the mask specified in the program memory just after the *jsr* instruction.

Returns

The resulting value is in long register.

See Also

c_bfput

c_bfput

Description

Store a long bitfield

Syntax

```
; value in long register
; pointer to bitfield in y register
    jsr   c_bfput
    dc.l  mask
; result in long register
```

Function

c_bfput is storing a long bitfield at the address loaded in the **Y** register by shifting and masking the long register from a mask specified in the program memory just after the *jsr* instruction.

Returns

Nothing.

See Also

c_bfget

c_check

Description

Check stack growth

Syntax

```
        leay #<size>,s
        jsr  c_check
```

Function

c_check is used to check that the stack pointer is not overwriting valid data in memory. Users must write their own check functions, because the memory map is application-dependent. The value in **Y** is the new stack pointer value.

Returns

c_check returns only if the stack pointer is correct. Otherwise, the behavior is user-dependent. *c_check* is called when the **-ck** flag is specified (raised) to the code generator (**cg6812**). This option produces larger and slower code. It should only be used for test and debugging. The libraries provided with the compiler include a version of *c_check* that always returns. It may be used as a template for user-written versions of this function.

c_dadd

Description

Add double to double

Syntax

```
; pointer to left in x register
; pointer to right in y register
    jsr   c_dadd
; result in left
```

Function

c_dadd adds the double in left to the double in right. No check is made for overflow.

Returns

The resulting value is in left. Flags have no meaningful value upon return.

See Also

c_dsub

c_dcmp

Description

Compare double with double

Syntax

```
; pointer to left in x register
; pointer to right in y register
     jsr   c_dcmp
; result in flags
```

Function

c_dcmp compares the double in left with the double in right.

Returns

The **N** and **Z** flags are set to reflect the value of (left-right).

c_ddiv

Description

Divide double by double

Syntax

```
; pointer to left in x register
; pointer to right in y register
     jsr   c_ddiv
; result in left
```

Function

c_ddiv divides the double in left by the double in right.

Returns

The resulting value is in left. A zero divide leaves the operand unchanged. Flags have no meaningful value upon return.

c_dmul

Description

Multiply double by double

Syntax

```
; pointer to left in x register
; pointer to right in y register
    jsr   c_dmul
; result in left
```

Function

c_dmul multiplies the double in left by the double in right.

Returns

The resulting value is in left. Flags have no meaningful value upon return.

c_dneg

Description

Negate a double

Syntax

```
; pointer to operand in y register
     jsr   c_dneg
; result in operand
```

Function

c_dneg negates the double pointed at by the **Y** register.

Returns

The result stays in operand. The flags are not significant on return.

c_dsmov

Description

Move a structure in **DPAGE** space

Syntax

```
; source address on the stack
; destination address in X:D
    ldy   #<size>
    jsr   c_dsmov
```

Function

c_dsmov moves a structure inside the **DPAGE** data space. Both source and destination addresses are far pointer, pointer to source is on the stack, and pointer to destination is in the register pair **X:D**. The structure size is in the **Y** register.

See Also

c_esmov

c_dsub

Description

Subtract double from double

Syntax

```
; pointer to left in x register
; pointer to right in y register
     jsr   c_dsub
; result in left
```

Function

c_dsub subtracts the double in right from the double in left. No check is made for overflow.

Returns

The resulting value is in left. Flags have no meaningful value upon return.

See Also

c_dadd

c_dtod

Description

Copy a double into a double

Syntax

```
; pointer to left in x register
; pointer to right in y register
     jsr   c_dtod
; result in left
```

Function

c_dtod copies the double in right to left.

Returns

The right value is in left. Flags have no meaningful value upon return.

c_dtof

Description

Convert double to float

Syntax

```
; pointer to double in y register
      jsr   c_dtof
; result in float register
```

Function

c_dtof converts the double pointed at by **Y** to a float in the float register. No check is made for overflow.

Returns

The resulting value is in the float register. Flags have no meaningful value upon return.

c_dtoi

Description

Convert double to integer

Syntax

```
; pointer to operand in y register
     jsr   c_dtoi
; result in d
```

Function

c_dtoi converts the double pointed at by **Y** to a two byte integer in **D**. No check is made for overflow.

Returns

The resulting value is in **D** and flags are set accordingly.

c_dtol

Description

Convert double into long integer

Syntax

```
; pointer to double in y register
     jsr   c_dtol
; result in long register
```

Function

c_dtol converts the double in memory pointed at by **Y** to a long in the long register. No check is made for overflow.

Returns

The resulting value is in the long register.

c_dtos

Description

Copy a double onto the stack

Syntax

```
  ; pointer to double in y register
       jsr   c_dtos
```

Function

c_dtos copies the double pointed to by **Y** onto the stack.

Returns

c_dtos returns nothing; the stack is updated.

c_eewbfb

Description

Eeprom char bit field update

Syntax

```
        ldb   #value
        ldy   #address
        jsr   c_eewbfb
        dc.b  <mask>
```

Function

c_eewbfb updates a char bit field (8 bits sized) located in *eeprom* with a new value. The new value is in **B** and is right justified. The byte address in *eeprom* is in **Y**, and the mask, giving the bit field size and location, is a byte located in memory just after the call. The function waits for the time necessary to program the new value.

See Also

c_eewbfd, c_eewbfx, c_eewstx, c_eewsty

c_eewbfd

Description

Eeprom short bit field update

Syntax

```
        ldd   #value
        ldy   #address
        jsr   c_eewbfd
        dc.w  <mask>
```

Function

c_eewbfd updates a short bit field (16 bits sized) located in *eeprom* with a new value. The new value is in **D** and is right justified. The word address in *eeprom* is in **Y**, and the mask, giving the bit field size and location, is a word located in memory just after the call. The function waits as required to program the new value.

See Also

c_eewbfb, c_eewbfx, c_eewstx, c_eewsty

c_eewbfx

Description

Eeprom long bit field update

Syntax

```
 ; value in long register
     ldy   #address
     jsr   c_eewbfx
     dc.w  <mask>
```

Function

c_eewbfx updates a long bit field (32 bits sized) located in *eeprom* with a new value. The new value is in **X:D** and is right justified. The long address in *eeprom* is in **Y**, and the mask, giving the bit field size and location, is a word located in memory just after the call. The function waits as required to program the new value.

See Also

c_eewbfb, *c_eewbfd*, *c_eewstx*, *c_eewsty*

c_eewra

Description

Write a short int aligned in *eeprom*

Syntax

```
        ldd   #value
        ldy   #address
        jsr   c_eewra
```

Function

c_eewra writes a short int in *eeprom*. The new value is in **D**, and its address in *eeprom* is in **Y**, and is assumed to be even allowing the full word to be programmed with one single cycle.

See Also

c_eewrc, c_eewrd, c_eewrl, c_eewrw

c_eewrc

Description

Write a char int in *eeprom*

Syntax

```
        ldb   #value
        ldy   #address
        jsr   c_eewrc
```

Function

c_eewrc writes a byte in *eeprom*. The new byte value is in **B** and its address in *eeprom* is in **Y**. The function tests if the erasure is necessary, and it performs only in that case. Then if the new value is different from the one in *eeprom*, the new byte is programmed. The function waits for the time necessary to correctly program the byte. The delay function included in this module assumes that the clock frequency is **8 Mhz**. The function does not test if the byte address is in the address range corresponding to the existing *eeprom*.

See Also

c_eewra, c_eewrd, c_eewrl, c_eewrw

c_eewrd

Description

Write a double in *eeprom*

Syntax

```
; pointer to destination in x register
; pointer to source in y register
     jsr  c_eewrd
```

Function

c_eewrd writes a double in *eeprom*. If the destination address is even, all words are programmed by the *c_eewra* function. Otherwise, the first and last bytes are programmed by the *c_eewrc* function, and the middle words are programmed by the *c_eewra* function. The function waits as required to program all the bytes.

See Also

c_eewra, *c_eewrc*, *c_eewrl*, *c_eewrw*

c_eewrl

Description

Write a long int in *eeprom*

Syntax

```
  ; value in long register
       ldy   #address
       jsr   c_eewrl
```

Function

c_eewrl writes a long int in *eeprom*. The new value is in the long register, and its address in *eeprom* is in **Y**. If the destination is even, each word is written by the *c_eewra* function. Otherwise, the first and last bytes are programmed independently by the *c_eewrc* function, and the middle word is programmed by the *c_eewra* function. The function waits as required to program all the bytes.

See Also

c_eewra, *c_eewrc*, *c_eewrd*, *c_eewrw*

c_eewrw

Description

Write a short int in *eeprom*

Syntax

```
        ldd   #value
        ldy   #address
        jsr   c_eewrw
```

Function

c_eewrw writes a short int in *eeprom*. The new value is in **D**, and its address in *eeprom* is in **Y**. If the destination address is even, the word is programmed directly by a single programming cycle. Otherwise, each byte is programmed independently by the *c_eewrc* function.

See Also

c_eewra, c_eewrc, c_eewrd, c_eewrl

c_eewstx

Description

Move a structure in *eeprom*

Syntax

```
        ldy   #source_address
        ldx   #destination_address
        ldd   #<size>
        jsr   c_eewstx
```

Function

c_eewstx moves a structure into an *eeprom* memory location.Pointer to source is in **Y**, and pointer to destination is in **X**. The structure size is given by a word located in the **D** register. Depending on the size and the destination address alignment, as many words as possible are programmed by the *c_eewra* function. Remainning bytes are programmed by the *c_eewrc* function.

See Also

c_eewbfb, c_eewbfd, c_eewbfx, c_eewra, c_eewrc, c_eewsty

c_eewsty

Description

Move a structure in *eeprom*

Syntax

```
        ldx   #source_address
        ldy   #destination_address
        ldd   #<size>
        jsr   c_eewsty
```

Function

c_eewsty moves a structure into an *eeprom* memory location.Pointer to source is in **X**, and pointer to destination is in **Y**. The structure size is given by a word located in the **D** register. Depending on the size and the destination address alignment, as many words as possible are programmed by the *c_eewra* function. Remainning bytes are programmed by the *c_eewrc* function.

See Also

c_eewbfb, c_eewbfd, c_eewbfx, c_eewra, c_eewrc, c_eewstx

c_emuld

Description

Multiply signed int by unsigned int

Syntax

```
; signed int in d register
; unsigned int in y register
     jsr   c_emuld
; long result in y:d register pair
```

Function

c_emuld multiplies the *signed* int value in the **D** register by the *unsigned* int value in the **Y** register. The 32 bits result is stored in the register pair **Y:D**.

See Also

c_emuly

c_emuly

Description

Multiply unsigned int by signed int

Syntax

```
; unsigned int in d register
; signed int in y register
    jsr   c_emuly
; long result in y:d register pair
```

Function

c_emuly multiplies the *unsigned* int value in the **D** register by the *signed* int value in the **Y** register. The 32 bits result is stored in the register pair **Y:D**.

See Also

c_emuld

c_esmov

Description

Move a structure in **EPAGE** space

Syntax

```
; source address on the stack
; destination address in x:d
    ldy   #<size>
    jsr   c_esmov
```

Function

c_esmov moves a structure inside the **EPAGE** data space. Both source and destination addresses are far pointer, pointer to source is on the stack, and pointer to destination is in the register pair **X:D**. The structure size is in the **Y** register.

See Also

c_dsmov

c_fadd

Description

Add float to float

Syntax

```
; left in float register
; pointer to right in y register
     jsr   c_fadd
; result in float register
```

Function

c_fadd adds the float in float register to the float indicated by the pointer in **Y**. No check is made for overflow.

Returns

The resulting value is in the 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 y register
     jsr   c_fcmp
; result in flags
```

Function

c_fcmp compares the float in the float register with the float pointed at by the **Y** register.

Returns

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 y register
      jsr   c_fdiv
; result in float register
```

Function

c_fdiv divides the float in the float register by the float pointed to by the **Y** register.

Returns

The resulting value is in the float register. Flags have no meaningful value upon return.

c_fgadd

Description

Float addition

Syntax

```
; pointer to left in y register
; right in float register
     jsr   c_fgadd
; result in left
```

Function

c_fgadd performs the float addition of the value pointed at by **Y** and the value in the float register.

Returns

The result is stored at the location pointed at by the **Y** register.

c_fgdiv

Description

Float division

Syntax

```
; pointer to left in y register
; right in float register
     jsr   c_fgdiv
; result in left
```

Function

c_fgdiv performs the float division of the value pointed at by the **Y** register by the value in float register.

Returns

The result is stored in the location pointed at by **Y**.

c_fgmul

Description

Float multiplication

Syntax

```
; pointer to left in y register
; right in float register
     jsr   c_fgmul
; result in left
```

Function

c_fgmul performs the float multiplication of the value pointed at by the **Y** register by the value in float register.

Returns

The result is stored in the location pointed at by **Y**.

c_fgsub

Description

Float subtraction

Syntax

```
; pointer to left in y register
; right operand in float register
     jsr   c_fgsub
; result in left
```

Function

c_fgsub evaluates the (float) difference between the value pointed at by the **Y** register and the value in float register

Returns

The result is stored in the location pointed at by **Y**.

c_fmul

Description

Multiply float by float

Syntax

```
; left in float register
; pointer to right in y register
     jsr   c_fmul
; result in float register
```

Function

c_fmul multiplies the float in the float register by the float pointed to by the **Y** register.

Returns

The resulting value is in the 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 y register
    jsr  c_fsub
; result in float register
```

Function

c_fsub subtracts the float pointed to by the **Y** register from the float in the float register. No check is made for overflow.

Returns

The resulting value is in the float register. Flags have no meaningful value upon return.

See Also

c_fadd

c_ftod

Description

Convert float into double

Syntax

```
  ; pointer to double in y register
  ; float in float register
       jsr   c_ftod
  ; result in memory
```

Function

c_ftod converts the float in the float register to a double in the memory pointed at by **Y**.

Returns

The resulting value is in memory at the location pointedt by the **Y** register. Flags have no meaningful value upon return.

c_ftoi

Description

Convert float to integer

Syntax

```
; float in float register
     jsr   c_ftoi
; result in d
```

Function

c_ftoi converts the float in the float register to a two byte integer in **D**. No check is made for overflow.

Returns

The resulting value is in **d**. Flags have no meaningful value upon return.

c_ftol

Description

Convert float into long integer

Syntax

```
; float in float register
      jsr   c_ftol
; result in long register
```

Function

c_ftol converts the float in the float register to a four byte integer in long register. No check is made for overflow.

Returns

The resulting value is in the long register. Flags have no meaningful value upon return.

c_itod

Description

Convert integer into double

Syntax

```
; pointer to double in y register
     ldd   value
     jsr   c_itod
; result in memory
```

Function

c_itod converts the two byte integer in **D** to a double stored in memory at the address specified by the **Y** register.

Returns

The resulting value is in memory at the address specified by the **Y** register. Flags have no meaningful value upon return.

c_itof

Description

Convert integer into float

Syntax

```
        ldd value
        jsr   c_itof
; result in float register
```

Function

c_itof converts the two byte integer in **D** to a float stored in the float register.

Returns

The resulting value is in the float register. Flags have no meaningful value upon return.

c_jltab

Description

Perform C switch statement on long

Syntax

```
; <value> in long register
      ldy   #swtab
      jsr   c_jltab
```

Function

c_jltab is called to switch to the proper code sequence, depending on a value and an address table. The top of the table is specified in the **Y** register, and consists of a list of ranges followed by a list of pairs. A range consists of a header, made of a count followed by a starting value, followed by an address list. A header with a zero count indicates the final list of pairs. The count is followed in this case by the number of following pairs. A pair consists of an address followed by a value. The pair list is ended by the default address. All values are four byte integers. All addresses are two byte integers.

Returns

c_jltab jumps to the proper code. It never returns.

c_jptab

Description

Perform C switch statement in PIC mode

Syntax

```
        ldd   value
        jsr   c_jptab,pcr
        <offsets>
```

Function

c_jptab is called to switch to the proper code sequence, depending on a value and an offset table. The top of the table is found on the stack after the function is entered, and consists of a list of offsets allowing the functions to compute the physical target address.

Returns

c_jptab jumps to the proper code. It never returns.

c_jtab

Description

Perform C switch statement

Syntax

```
        ldd   value
        ldy   #swtab
        jsr   c_jtab
```

Function

c_jtab is called to switch to the proper code sequence, depending on a value and an address table. The top of the table is specified in the **Y** register, and consists of a list of ranges followed by a list of pairs. A range consists of a header, made of a count followed by a starting value, followed by an address list. A header with a zero count indicates the final list of pairs. The count is followed in this case by the number of following pairs. A pair consists of an address followed by a value. The pair list is ended by the default address. All values and addresses are two byte integers.

Returns

c_jtab jumps to the proper code. It never returns.

c_ladd

Description

Long integer addition

Syntax

```
 ; left in long register
 ; pointer to right in y register
        jsr   c_ladd
 ; result in long register
```

Function

c_ladd adds the four byte integer, left and the four byte integer, right.

Returns

The result is in left. Flags are not significant on return.

See Also

c_lcmp, c_lsub

c_land

Description

Bitwise AND for long integers

Syntax

```
; left in long register
; pointer to right in y register
    jsr  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.

Returns

The result is in the long register. Flags are not significant on return.

See Also

c_lor, c_lxor

c_lcmp

Description

Long integer compare

Syntax

```
; left in long register
; pointer to right in y register
    jsr   c_lcmp
; result in flags
```

Function

c_lcmp compares the four byte integer, left with the four byte integer pointed at by **Y**.

Returns

Flags are set accordingly.

See Also

c_ladd, c_lsub, c_pcmp

c_ldiv

Description

Quotient of long integer division

Syntax

```
  ; left in long register
  ; pointer to right in y register
       jsr   c_ldiv
  ; quotient in long register
```

Function

c_ldiv divides the four byte integer in the long register by the four byte integer pointed at by **Y**. Values are assumed to be signed. If division by zero is attempted, results are as provided by the divide instruction.

Return

The quotient is in the long register; The flags are not significant on return.

See Also

c_ludv, c_lmod, c_umd

c_lgadd

Description

Long addition

Syntax

```
    ; pointer to left in y register
    ; right in long register
         jsr   c_lgadd
    ; result in left
```

Function

c_lgadd performs the long addition of the value pointed at by **Y** and the value in the long register.

Returns

The result is stored at the location pointed at by the **Y** register.

c_lgand

Description

Long bitwise AND

Syntax

```
; pointer to left in y register
; right in long register
     jsr  c_lgand
; result in memory (left is updated)
```

Function

c_lgand performs the long bitwise AND of the value pointed at by the **Y** register and the value in the long register.

Returns

The results is stored at the location pointed at by the **Y** register, meaning that the left operand is updated.

c_lgdiv

Description

Quotient of long division

Syntax

```
; pointer to left in y register
; right in long register
     jsr   c_lgdiv
; result in left
```

Function

c_lgdiv performs the long division of the value pointed at by the **Y** register by the value in long register.

Returns

The result is stored in the location pointed at by **Y**.

c_lglsh

Description

Long shift left

Syntax

```
; pointer to long in y register
; shift count in d register
     jsr  c_lglsh
; result in memory
```

Function

c_lglsh performs the long left shift of the value pointed at by the **Y** register, by the bit count in **D**. No check is done against silly counts.

Returns

The result is stored in the location pointed at by **Y**.

c_lgmod

Description

Remainder of long division

Syntax

```
; pointer to left in y register
; right in long register
     jsr   c_lgmod
; result in left
```

Function

c_lgmod performs the long division of the value pointed at by the **Y** register by the value in long register and stores the remainder.

Returns

The result is stored in the location pointed at by **Y**.

c_lgmul

Description
Long multiplication

Syntax
```
; pointer to left in y register
; right in long register
    jsr   c_lgmul
; result in left
```

Function
c_lgmul performs the long multiplication of the value pointed at by the **Y** register by the value in long register.

Returns
The result is stored in the location pointed at by **Y**.

c_lgor

Description

Long bitwise OR

Syntax

```
; pointer to left in y register
; right operand in long register
     jsr   c_lgor
; result in left
```

Function

c_lgor performs the long bitwise OR of the value pointed at by **Y** and the value in the long register.

c_lgrsh

Description

Signed long shift right

Syntax

```
; pointer to long in y register
; shift count in d register
     jsr   c_lgrsh
; result in memory
```

Function

c_lgrsh performs the signed long right shift of the value pointed at by the **Y** register, by the bit count in the **D** register. No check is done against silly counts. Because the value is signed, arithmetic shift instructions are used.

Returns

The result is stored in the location pointed at by **y**.

c_lgudv

Description

Quotient of unsigned long division

Syntax

```
 ; pointer to left in y register
 ; right in long register
      jsr   c_lgudv
 ; result in left
```

Function

c_lgudv performs the unsigned long division of the value pointed at by the **Y** register by the value in long register.

Returns

The result is stored in the location pointed at by **Y**.

c_lgumd

Description

Remainder of unsigned long division

Syntax

```
; pointer to left in y register
; right in long register
      jsr   c_lgumd
; result in left
```

Function

c_lgumd performs the unsigned long division of the value pointed at by the **Y** register by the value in long register and stores the remainder.

Returns

The result is stored in the location pointed at by **Y**.

c_lgursh

Description

Unsigned long shift right

Syntax

```
; pointer to long in y register
; shift count in d register
    jsr   c_lgursh
; result in memory
```

Function

c_lgursh performs the unsigned long right shift of the value pointed at by the **Y** register, by the bit count in the **D** register. No check is done against silly counts. Because the value is unsigned, logical shift instructions are used.

Returns

The result is stored in the location pointed at by **Y**.

c_lgsub

Description

Long subtraction

Syntax

```
; pointer to left in y register
; right operand in long register
    jsr   c_lgsub
; result in left
```

Function

c_lgsub evaluates the (long) difference between the value pointed at by the **Y** register and the value in long register

Returns

The result is stored in the location pointed at by **Y**.

c_lgxor

Description

Long bitwise exclusive OR

Syntax

```
; pointer to left in y register
; right operand in long register
     jsr   c_lgxor
; result in left
```

Function

c_lgxor performs the long bitwise EXOR (exclusive OR) of the value pointed at by the **Y** register and the value in long register.

Returns

The result is stored in the location pointed at by **Y**.

c_llsh

Description
Long shift left

Syntax
```
  ; operand in long register
  ; shift count in y register
       jsr   c_llsh
  ; result in long register
```

Function
c_llsh performs the long left shift of the value in the long register, by the bit count in **Y**. No check is done against silly counts.

Returns
The result is in the long register.

c_lmod

Description

Remainder of long integer division

Syntax

```
 ; left in long register
 ; pointer to right in y register
        jsr   c_lmod
 ; remainder in long register
```

Function

c_lmod divides the four byte integer in long register by the four byte integer pointed at by the **Y** register. Values are assumed to be signed.

Returns

The remainder appears in the long register.

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 y register
     jsr   c_lmul
; result in long register
```

Function

c_lmul multiplies the four byte integer in the long register by the four byte integer pointed at by the **Y** register. No check is made for overflow.

Returns

The resulting value is in the long register.

c_lneg

Description

Negate a long integer

Syntax

```
; value in long register
      jsr   c_lneg
; result in long register
```

Function

c_lneg negates the four byte integer in the long register.

Returns

The result stays in the long register. The flags are not significant on return.

See Also

c_lcom

c_lor

Description

Bitwise OR with long integers

Syntax

```
; left in long register
; pointer to right in y register
        jsr   c_lor
; result in long register
```

Function

c_lor operates a bitwise OR between the contents of the long register and the long pointed at by the **Y** register. Each operand is taken to be a four byte integer.

Returns

The result is in the long register. The flags are not significant on return.

See Also

c_land, *c_lxor*

c_lrsh

Description

Signed long shift right

Syntax

```
; operand in long register
; shift count in y register
     jsr   c_lrsh
; result in long register
```

Function

c_lrsh performs the signed long right shift of the value in the long register, by the bit count in the **Y** register. No check is done against silly counts. Because the value is signed, arithmetic shift instructions are used.

Returns

The result is in the long register.

c_lrzmp

Description

Long test against zero

Syntax

```
; value in long register
      jsr   c_lrzmp
; result in the flags
```

Function

c_lrzmp tests the value in the long register and updates the sign and zero flags.

Returns

Nothing, but the (possibly changed) flags.

c_lsub

Description

Long integer subtraction

Syntax

```
  ; long in long register
  ; pointer to right in y register
        jsr   c_lsub
  ; result in long register
```

Function

c_lsub subtracts the four byte integer pointed at by the **Y** register from the four byte integer in the long register.

Returns

The result is in the long register. Flags are not significant on return.

See Also

c_ladd, c_lcmp

c_ltod

Description

Convert long integer into double

Syntax

```
; pointer to double in y register
; long in long register
    jsr   c_ltod
; result in memory
```

Function

c_ltod converts the four byte integer in the long register to a double pointed at by the **Y** register.

Returns

The resulting value is in memory. Flags have no meaningful value upon return.

c_ltof

Description

Convert long integer into float

Syntax

```
; value in long register
      jsr   c_ltof
; result in float register
```

Function

c_ltof converts the four byte integer in the long register to a float.

Returns

The resulting value is in the float register. Flags have no meaningful value upon return.

c_ludv

Description

Quotient of unsigned long integer division

Syntax

```
; left in long register
; pointer to right in y register
     jsr   c_lduv
; quotient in long register
```

Function

c_ludv divides the four byte integer in the long register by the four byte integer pointed at by the **Y** register. Values are assumed to be unsigned.

Returns

The quotient is in the long register. The flags are not significant on 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 y register
        jsr   c_lumd
   ; remainder in long register
```

Function

c_lumd divides the four byte integer in the long register by the four byte integer pointed at by the **Y** register. Values are assumed to be unsigned.

Returns

The remainder is in the long register. The flags are not significant on return.

See Also

c_lmod, c_ldiv, c_ludv

c_lursh

Description

Unsigned long shift right

Syntax

```
  ; operand in long register
  ; shift count in y register
        jsr   c_lursh
  ; result in long register
```

Function

c_lursh performs the unsigned long right shift of the value in the long register, by the bit count in the **Y** register. No check is done against silly counts. Because the value is unsigned, logical shift instructions are used.

Returns

The result is in the long register.

c_lursha

Description

Truncated unsigned long shift right

Syntax

```
; operand in b and x register
; shift count in a register
    jsr   c_lursha
; result in long register
```

Function

*c_lursh*a performs the unsigned right shift of the value in the **B** and **X** registers, by the bit count in the **A** register. It is used to shift a long integer in memory by a count greater than 8 by loading the 3 most significant bytes in the **B** and **X** registers. Because the value is unsigned, logical shift instructions are used.

Returns

The result is in the long register.

c_lxor

Description

Bitwise exclusive OR with long integers

Syntax

```
 ; left in long register
 ; pointer to right in y register
       jsr   c_lxor
 ; result in long result
```

Function

c_lxor operates a bitwise exclusive OR between the contents of the long register and the long pointed at by the **Y** register. Each operand is taken to be a four byte integer.

Returns

The result is in the long register. The flags are not significant on return.

See Also

c_land, c_lor

c_lzmp

Description

Compare a long integer to zero

Syntax

```
; value in long register
      jsr   c_lzmp
; result in the flags
```

Function

c_lzmp compares the four byte integer in the long register with zero.

Returns

Nothing. The **Z** flags is updated.

c_pcmp

Description

Far pointer compare

Syntax

```
; left in far pointer register
; pointer to right in y register
     jsr  c_pcmd
; result in flags
```

Function

c_pcmp compares the three byte pointer, left with the three byte pointer pointed at by **Y**.

Returns

Flags are set accordingly.

See Also

c_lcmp

c_uitod

Description

Convert unsigned integer into double

Syntax

```
; pointer to double in y register
       ldd   value
       jsr   c_uitod
; result in memory
```

Function

c_uitod converts the two byte unsigned integer in **D** to a double stored in memory, and pointed at by the **Y** register.

Returns

The resulting value is in memory at the address specified by the **Y** register. Flags have no meaningful value upon return.

c_uitof

Description

Convert unsigned integer into float

Syntax

```
        ldd   value
        jsr   c_uitof
  ; result in float register
```

Function

c_uitof converts the two byte unsigned integer in **D** to a float stored in the float register.

Returns

The resulting value is in the float register. Flags have no meaningful value upon return.

c_ultod

Description

Convert unsigned long integer into double

Syntax

```
; pointer to double in y register
; long in long register
     jsr   c_ultod
; result in memory
```

Function

c_ultod converts the four byte unsigned integer in the long register to a double pointed at by the **Y** register.

Returns

The resulting value is in memory. Flags have no meaningful value upon return.

c_ultof

Description

Convert unsigned long integer into float

Syntax

```
  ; value in long register
        jsr   c_ultof
  ; result in float register
```

Function

c_ultof converts the four byte unsigned integer in the long register to a float.

Returns

The resulting value is in the float register. Flags have no meaningful value upon return.

APPENDIX

D

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 **cx6812** compiler uses to control the compilation process.

This appendix describes each of the passes of the compiler:

cp6812	the parser
cg6812	the code generator
co6812	the assembly language optimizer

The cp6812 Parser

cp6812 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 **cg6812.**

Command Line Options

cp6812 accepts the following options, each of which is described in detail below:

```
cp6812 [options] file
       -ad      expand defines in assembly
       -b       even align all data
       -bc      even align constants
       -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
       -f       single precision floats
       -ge      eclipse error messages
       -h*>     include header
       -i*>     include path
       -kc      keep space on init cast
       -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
       -nw      do not widen arguments
       -o*      output file name
       -p       need prototypes
       -rb      reverse bitfield order
       -s       do not reorder locals
       -sa      strict ANSI conformance
       -u       plain char is unsigned
       -wp      warn unknown pragmas
       -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
-ad	enable #define expansion inside inline assembly code between #asm and #endasm directives. By default, #define symbols are expanded only in the C code.
-b	enforce storage boundaries to begin on an even bound.
-bc	enforce const variable storage boundaries to begin on an even bound.
-c99	enable the supported C99 extensions.
-cc	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 long instead of the default type int. Such expressions were allowing intermediate results to become larger that an int without any truncation.
-ck	enable extra type checking. For more information, see "*Extra verifications*" below.
-cp	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 ...).
-csb	produce an error message if a bitfield is declared explicitly with the **signed** keyword. By default, the compiler silently ignores the signed feature and handles all bitfields as unsigned values.
-d*^	specify ***** as the name of a user-defined preprocessor symbol (**#define**). The form of the definition is **-dsymbol[=value]**; the symbol is set to **1** if value is omitted. You can specify up to 128 such definitions.
-e	run preprocessor only. *cp6812* only outputs lines of text.
+e*	log errors in the text file ***** instead of displaying the messages on the terminal screen.

Parser Option Usage (cont.)

Option	Description
-f	treat all floating point numbers as float and not double, even if they are declared as double. All calculations will be made on 32 bits instead of 64 bits. Space reservations will be made on a 32 bit basis, as well as argument passing.
-ge	produce error messages directly compatible with the Eclipse environment
-h*>	include files before to start the compiler process. You can specify up to 128 files.
-i*>	specify include path. You can specify up to 128 different paths. Each path is a directory name, **not** terminated by any directory separator character, or a file containing an unlimited list of directory names.
-kc	keep the original pointer space when casting it to a pointer in the initialization part of a declaration.
-ku	keep unused statics. By default, unused statics are removed.
-l	output line number information for listing or debug.
-md	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.

Parser Option Usage (cont.)

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>-m#</td><td>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:

<table>
<tr><td>0x10</td><td>@tiny</td></tr>
<tr><td>0x20</td><td>@near</td></tr>
<tr><td>0x30</td><td>@far</td></tr>
</table>

Allowed optional specifiers are:

<table>
<tr><td>0x01</td><td>@pack</td></tr>
<tr><td>0x04</td><td>@nostack</td></tr>
</table>

Note that all the combinations are not significant for all the target processors.</td></tr>
<tr><td>-nb</td><td>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.</td></tr>
<tr><td>-nc</td><td>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.</td></tr>
<tr><td>-ne</td><td>do not optimize size of enum variables. By default, the compiler selects the smallest integer type by checking the range of the declared enum members. This mechanism does not allow incomplete enum declaration. When the -ne option is selected, all enum variables are allocated as int variables, thus allowing incomplete declarations, as the knowledge of all the members is no more necessary to choose the proper integer type.</td></tr>
</table>

Parser Option Usage (cont.)

Option	Description
-np	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.
-ns	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.
-nw	do not widen arguments. The standard behaviour of the compiler is to widen integer arguments smaller than **int** to **int size** and to widen **float** arguments to **double**. If this flag is set, these promotions are not done. The code thus obtained should be more compact if **char** and **floats** are heavily used.
-o*	write the output to the file *. Default is STDOUT for output if **-e** is specified. Otherwise, an output file name is required.
-p	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.
-rb	reverse the bitfield fill order. By default, bitfields are filled from **less** significant bit (LSB) to **most** significant bit (MSB). If this option is specified, filling works from most significant bit to less significant bit.
-s	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.
-sa	enforce a strict ANSI checking by rejecting any syntax or semantic extension. This option also disables the enum size optimization (**-ne**).
-u	take a plain char to be of type **unsigned char**, not signed char. This also affects in the same way strings constants.

Parser Option Usage (cont.)

Option	Description
-wp	produce a warning for any unknown #pragma directives. By default, unknown #pragma directives are silently ignored as requested by the ANSI/ISO standard.
-w#	enable warnings from level #. By default, warnings are disabled. For more information, see "*Warning Levels*" below.
-x	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.
-xd	add debug information in the object file only for data objects, hiding any function.
-xf	prefix filenames in the debug information with absolute full path name.
-xp	do not prefix filenames in the debug information with any absolute path name. Debuggers will have to be informed about the actual files location.
-xu	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.
-xx	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 beeing initialized by an explicit assignment.

Return Status

cp6812 returns success if it produces no error diagnostics.

Example

cp6812 is usually invoked before *cg6812* the code generator, as in:

```
cp6812 -o \2.cx1 -u -i \cosmic\h6812 file.c
cg6812 -o \2.cx2 \2.cx1
```

The cg6812 Code Generator

cg6812 is the code generating pass of the C compiler. It accepts a sequential file of flow graphs and parse trees from *cp6812* 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

cg6812 accepts the following options, each of which is described in detail below:

```
cg6812 [options] file
       -a      optimize _asm code
       -bss    do not use bss
       -bv     volatile bitfields access
       -cf     far constants mode
       -ck     check stack frame
       -cr#    position independent code
       -cs     split constants section
       -ct     constants in code
       -df     far data not splitted
       -dl#    output line information
       -dv     inline ediv instruction
       +e*     error file name
       -f      full listing display
       -ge     eclipse error messages
       -i      inline machine calls
       -l      output listing
       -na     do not xdef alias name
       -no     do not use optimizer
       -o*     output file name
       -p      emit padding code
       -r*     registers base address
       -sf     split function sections
       -t#     processor type
       -tf     far function not separated
       -v      verbose
```

Code generator Option Usage

Option	Description
-a	optimize *_asm* code. By default, the assembly code inserted by a *_asm* call is left unchanged by the optimizer.
-bss	inhibit generating code into the *bss* section.
-bv	allow volatile bitfields accesses to be implemented by a *bclr/bset* instruction pair. By default, the compiler implements a construct which writes only once into the resulting byte to avoid unexpected transitory states on an output peripheral port. The default code is safer but produces a larger code than with the **-bv** option.
-cf	use **ppage** for **@far** const pointers. Default is to use **ppage** only for when **@far @ppage** pointers.
-ck	enable stack overflow checking. The compiler calls at the beginning of each C function the machine library routine *c_check*, setting a register to the expected value of the stack pointer. A template of this routine is provided in the machine library sources and must be completed to check this value against the stack limit and to decide what to do if an overflow is detected.
-cr#	produce **P**osition **I**ndependent **C**ode using the **pc** relative addressing modes both for function calls and constant data access. **#** must be either: **1**: if **+pic** option specified **7**: if **+picd** option specified **3**: if **+picds** option specified
-cs	split the **const** section into two sections. One for single byte constants and another for the rest so that can be allocated separately to avoid odd accesses.
-ct	output constant in the **.text** section. By default, the compiler outputs literals and constants in the **.const** section.
-df	do not use the **.fdata** section for **@far** variables, using instead the same allocation mechanism as plain data.

Code generator Option Usage (cont.)

Option	Description
-dl#	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 **-dl1**; *2)* file name and line number is obtained by specifying **-dl2**. All information is coded in symbols that are in the debug symbol table.
-dv	inline *ediv* instruction in constructs dividing long by a short or shifting right a long by up to sixteen bits.
+e*	log errors in the text file * instead of displaying the messages on the terminal screen.
-f	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.
-ge	produce error messages directly compatible with the Eclipse environment
-i	produce faster code by inlining machine library calls for long integers handling. The code produced will be larger than without this option.
-l	merge C source listing with assembly language code; listing output defaults to <*file*>.*ls*.
-na	do not produce an *xdef* directive for the *equate* names created for each C object declared with an absolute address.
-no	do not produce special directives for the post-optimizer.
-o*	write the output to the file * and write error messages to STDOUT. The default is STDOUT for output and STDERR for error messages.
-r*	define the I/O registers base address to allow the code generator to access the **PPAGE**, **DPAGE** and **EPAGE** registers directly, if no specific header file has been included. The compiler locates those registers in page 0 by default and uses the **-t** option value to select the proper addresses. The **-r** operand can be an absolute value or a symbol name which must be defined somewhere in the application.

Code generator Option Usage (cont.)

Option	Description
-sf	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.
-t#	specify the processor family where # may take values: **0** for the **68HC12A4**, **1** for the early **DG128** family, **2** for the **S12** family. This option allows the compiler to use the proper default addresses for the **PPAGE**, **DPAGE** and **EPAGE** registers if no specific header file was included. These addresses are updated according to the **-r** option.
-tf	if set, functions are generated in *.text* section. By default, they are generated in the *.ftext* section.
-v	When this option is set, each function name is send to STDERR when *cg6812* starts processing it.

Return Status

cg6812 returns success if it produces no diagnostics.

Example

cg6812 usually follows *cp6812* as follows:

```
cp6812 -o \2.cx1 -u -i\cosmic\h6812 file.c
cg6812 -o \2.cx2 \2.cx1
```


The co6812 Assembly Language Optimizer

co6812 is the code optimizing pass of the C compiler. It reads source files of HC12/HCS12 assembly language source code, as generated by the *cg6812* code generator, and writes assembly language statements. *co6812* is a peephole optimizer; it works by checking lines function by function for specific patterns. If the patterns are present, *co6812* 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

co6812 accepts the following options, each of which is described in detail below:

```
co6812 [options] <file>
     -c      keep original lines as comments
     -d*     disable specific optimizations
     -o*     output file name
     -v      print efficiency statistics
```

Optimizer Option Usage

Option	Description
-c	leave removed instructions as comments in the output file.
-d*	specify a list of codes allowing specific optimizations functions to be selectively disabled.
-o*	write the output to the file * and write error messages to STDOUT. The default is STDOUT for output and STDERR for error messages.
-v	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

co6812 returns success if it produces no diagnostics.

Example

co6812 is usually invoked after *cg6812* as follows:

```
cp6812 -o \2.cx1 -u -i\cosmic\h6812 file.c
cg6812 -o \2.cx2 \2.cx1
co6812 -o file.s \2.cx2
```

Index

Symbols

Numerics

A

B

C

D

E

H

I

L

M

T

U

V

W

X

Z