Calypsi C compiler guide for the 68000

Release 5.18

Håkan Thörngren

Jul 15, 2026

Contents

1 Introduction 1
- 1.1 Using this guide . . . 1
- 1.2 Copyright notice . . . 1
- 1.3 Disclaimer . . . 2

2 Installation 3
- 2.1 macOS . . . 3
- 2.2 Debian Linux . . . 4
- 2.3 Fedora Linux . . . 4
- 2.4 Arch and Manjaro Linux . . . 5
- 2.5 Windows . . . 5
- 2.6 Using multiple versions . . . 6

3 Tutorial 7
- 3.1 Simulator . . . 7
- 3.2 Amiga . . . 8
- 3.3 A2560 Foenix . . . 9
- 3.4 The Morfe emulator . . . 10

4 Getting started 13
- 4.1 C language . . . 13
- 4.2 File extensions . . . 14
- 4.3 Building applications . . . 14
- 4.4 Configuring . . . 15
- 4.5 Low level control . . . 17

5 Data storage 19
- 5.1 Ways to store data . . . 19
- 5.2 Address spaces . . . 20

6 Placing code and data 23
- 6.1 Sections . . . 23
- 6.2 Data initialization . . . 25
- 6.3 Stack . . . 26

7 C language 27
- 7.1 Overview . . . 27

7.2 Extensions . . . 27

8 Efficient coding 29
8.1 Data types . . . 29
8.2 Use prototypes . . . 30
8.3 Use tables . . . 31

9 Target specifics 33
9.1 Amiga . . . 33
9.2 Foenix A2560 . . . 34

10 Running the compiler 35
10.1 Basic invocation . . . 35
10.2 Include search path . . . 36
10.3 Compiler output . . . 36
10.4 Diagnostics . . . 38
10.5 Command line options . . . 39

11 Data representation 49
11.1 Basic types . . . 49
11.2 Structure types . . . 52
11.3 Union types . . . 52
11.4 Enumeration types . . . 53
11.5 Type qualifiers . . . 53
11.6 Type definitions . . . 56
11.7 Alignment . . . 56

12 Extended attributes 57
12.1 Overview . . . 57
12.2 Using attributes . . . 57
12.3 Attribute reference . . . 58

13 Language extensions 63
13.1 Overloaded functions . . . 63
13.2 Statement expressions . . . 63
13.3 Auto type . . . 64
13.4 Typeof . . . 65
13.5 Generics . . . 65

14 Pragma directives 67
14.1 Pragma directives reference . . . 67

15 Intrinsic functions 71
15.1 Intrinsic functions reference . . . 71

16 Preprocessor 75
16.1 Overview . . . 75

17 Section reference 81
17.1 Section overview . . . 81
17.2 Sections used by the compiler . . . 82

18 Runtime library 87
18.1 Design considerations . . . 87
18.2 Using the library . . . 88
18.3 Provided library files . . . 88

18.4 Stubs interface . . . 89
18.5 Examine use of library . . . 94
18.6 Library on a diet . . . 94
18.7 Override library functions . . . 96
18.8 C startup . . . 96
18.9 Time and date . . . 97
18.10 Rebuilding the C library . . . 98

19 The optimizer 99
19.1 Overview . . . 99
19.2 Function inlining . . . 100
19.3 Cross call . . . 103

20 Assembly language interface 105
20.1 Intrinsic functions . . . 105
20.2 Assembly functions . . . 105
20.3 Calling convention . . . 107
20.4 Inline assembler . . . 109

21 Assembler 113
21.1 Overview . . . 113
21.2 Syntax . . . 113
21.3 Sections . . . 114
21.4 Expressions . . . 116
21.5 Numeric constants . . . 116
21.6 Location counter . . . 117
21.7 Local labels . . . 117
21.8 Directives . . . 118
21.9 Relocations . . . 123
21.10 Macro language . . . 126

22 Running the assembler 129
22.1 Basic invocation . . . 129
22.2 Include search path . . . 130
22.3 Assembler output . . . 130
22.4 Command line options . . . 133

23 Linker 137
23.1 Running the linker . . . 137
23.2 Linking process . . . 138
23.3 Simplified placement rules . . . 138
23.4 ROM based systems . . . 139
23.5 Host based systems . . . 140
23.6 Objects . . . 140
23.7 Memory . . . 142
23.8 Linker rules file . . . 143
23.9 Scatter . . . 146
23.10 Linker list files . . . 147
23.11 Output formats . . . 151
23.12 More on linker rules . . . 152
23.13 Command line options . . . 152

24 Librarian 159
24.1 Creating a library . . . 159
24.2 File format . . . 159

24.3 Command line options . . . 159

25 Debugger introduction 161
25.1 Command line debugger . . . 161
25.2 Running the debugger . . . 162
25.3 Command line options . . . 162

26 Console command line 167
26.1 Command structure . . . 167
26.2 A simple debugging session . . . 168

27 Remote debugging 173
27.1 Serial port . . . 173

28 Visual Studio Code 175
28.1 Installation . . . 176
28.2 Project setup . . . 177
28.3 Building . . . 177
28.4 Running the debugger . . . 177

29 Lua scripting 181
29.1 Running a script . . . 181
29.2 Lua environments . . . 182
29.3 Command interpreters . . . 182
29.4 Console commands . . . 182
29.5 MI commands . . . 183
29.6 Adapted MI commands . . . 183
29.7 Choosing a command set . . . 183
29.8 Arrays and Lua . . . 183
29.9 Debugger API . . . 183

30 Lua data types reference 185
30.1 Address . . . 185
30.2 Adapted MI functions . . . 186

31 Lua function reference 187
31.1 Execution . . . 187
31.2 Breakpoint . . . 187

Index 189

CHAPTER 1

Introduction

Welcome to the Calypsi C compiler tool chain for the 68000. This guide provides reference information to help you optimize your use of the compiler toolchain suite. It also offers coding technique suggestions and insights into design and implementation choices made for this C compiler.

1.1 Using this guide

This guide is aimed for those programming the 68000 and provides in depth reference information on how to use the the Calypsi C compiler tool chain.

You should have a working knowledge of:

- The 68000 architecture
- The C programming language
- The operating system of your computer
- The text editor or IDE of your choice

1.2 Copyright notice

Copyright Håkan Thörngren.

This document is under copyright and may not be reproduced without the written consent from Håkan Thörngren.

The Calypsi C compiler tool chain is licensed and must be used according to its terms.

The C library runtime is based on NuttX, which is under the Apache license. The license can be found in `"Licenses/C Library License"` within the installation directory. Assembly-written runtime parts are covered by the the Calypsi C compiler tool chain license, imposing no royalties or obligations when used with this product.

The Calypsi C compiler tool chain internally uses several open source components. Their licenses are in the `"Licenses/Internal component licenses/"` directory within the installation directory. These licenses do not impose obligations on any application built with the Calypsi C compiler tool chain.

The 64-bit floating-point library routines are implemented using the Berkeley SoftFloat library. Its license is found in `Licenses/Berkeley SoftFloat License` in the installation directory. This license applies only when using the `long double` type or the `--64bit-doubles` command-line option. It carries the obligation to include its copyright and license with any product that uses the Berkeley SoftFloat library.

Note: The runtime library does not contain any GPL licensed code and does not impose any royalty. Some components as outlined above do carry some obligations, e.g. to mention that they are included in a final product.

1.3 Disclaimer

What is described in this document is subject to change without notice and does not represent any commitments. While what is described in this document is believed to be accurate, Håkan Thörngren takes no responsibility for any errors or omissions.

In no event shall Håkan Thörngren, its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

CHAPTER 2

Installation

This section contains installation information for supported platforms.

2.1 macOS

The macOS installer places the tools under `/usr/local`. Simply double-click the installation package to install the software.

After installation, files are in `/usr/local/calypsi-68000-5.18`, where `5.18` is the installed version. Each the Calypsi C compiler tool chain release creates a new directory, preventing overwrites of previous installations.

To simplify tool invocation, the installer script adds symbolic links from `/usr/local/bin` to the installation directory. Ensure `/usr/local/bin` is in your `PATH` environment variable to use the most recent installed version of the Calypsi C compiler tool chain. The installer also creates a symbolic link from `/usr/local/lib/calypsi-68000` to the installation directory, making it easy to refer to the installation without a version number.

2.1.1 Setting the PATH

To set the `PATH` environment variable on macOS, edit `/etc/paths` and add `/usr/local/bin`.

2.1.2 Uninstalling

To uninstall the Calypsi C compiler tool chain, run the uninstall script:

```
$ sudo /usr/local/lib/calypsi-68000-5.18/uninstall
```

This will remove the installation and the links in `/usr/local/bin`.

2.2 Debian Linux

The Debian package (`.deb` extension) is built and tested on Ubuntu 20.04. Installation on other Debian-based systems may be possible, but remains untested.

The package can be installed from the command line using:

```
$ sudo gdebi calypsi-68000-5.18.deb
```

Note: Download the Debian package to a local file before installation. `gdebi` does not fetch packages from a server.

If you do not have `gdebi` installed already, install it using:

```
$ sudo apt install gdebi-core
```

This installs the Calypsi C compiler tool chain in `/usr/local/lib/calypsi-68000-5.18`, where `5.18` is the installed version number.

To simplify tool invocation, the installer script adds symbolic links from `/usr/local/bin` to the installation directory. Ensure `/usr/local/bin` is in your `PATH` to use the latest the Calypsi tool chain. The installer also creates a symbolic link from `/usr/local/lib/calypsi-68000` to the installation directory. This allows referring to the installation without a version number.

It is also possible to install using `dpkg` in the following way:

```
$ sudo dpkg -i calypsi-68000-5.18.deb
```

This may fail if package dependencies are missing. The output will advise on necessary steps before retrying installation.

After installing any requested dependencies, retry the sudo dpkg installation command.

2.2.1 Uninstalling

To uninstall the Calypsi C compiler tool chain, use `dpkg` with the `-r` flag:

```
$ sudo dpkg -r calypsi-68000
```

2.3 Fedora Linux

The Fedora package (`.rpm` extension) is built and tested with Fedora version 40. Installation on other `.rpm` (Red Hat) based Linux systems may be possible, but remains untested.

The package file can be installed from the command line using:

```
$ sudo dnf install calypsi-68000-5.18-1.x86_64.rpm
```

Note: Download the package file to your local file system before installation.

This installs the Calypsi C compiler tool chain in `/usr/local/lib/calypsi-68000-5.18`, where `5.18` is the installed version number.

To simplify tool invocation, the installer script adds symbolic links from `/usr/local/bin` to the installation directory. Ensure `/usr/local/bin` is in your `PATH` to use the latest the Calypsi tool chain. The installer also creates a symbolic link from `/usr/local/lib/calypsi-68000` to the installation directory. This allows referring to the installation without a version number.

2.3.1 Uninstalling

To uninstall the Calypsi C compiler tool chain, use:

```
$ sudo dnf remove calypsi-68000-5.18-1.x86_64
```

2.4 Arch and Manjaro Linux

Manjaro is based on Arch Linux and uses `pacman`. Install the package file from the command line using:

```
$ sudo pacman -U calypsi-68000-5.18-x86_64.pkg.tar.zst
```

Note: Download the package file to your local file system before installation. `pacman -U` does not fetch packages from a server.

This installs the Calypsi C compiler tool chain in `/usr/local/lib/calypsi-68000-5.18`, where `5.18` is the installed version number.

To simplify tool invocation, the installer script adds symbolic links from `/usr/local/bin` to the installation directory. Ensure `/usr/local/bin` is in your `PATH` to use the latest the Calypsi tool chain. The installer also creates a symbolic link from `/usr/local/lib/calypsi-68000` to the installation directory. This allows referring to the installation without a version number.

2.4.1 Uninstalling

To uninstall the Calypsi C compiler tool chain, use `pacman` with the `-R` flag:

```
$ sudo pacman -R calypsi-68000
```

2.5 Windows

Extract the Windows product `calypsi-68000-5.18.zip` to a suitable location. It will expand into a folder `calypsi-68000-5.18`, allowing easy management of multiple versions.

2.5.1 Setting the PATH

To use the tools, add the bin folder inside the extracted archive to your system's search path. Search for "path" in the Start menu, then select "Edit the system environment variables - control panel".

2.5.2 Uninstalling

Uninstall by deleting the extracted archive via file explorer or command line.

2.6 Using multiple versions

On Linux, upgrading the software removes previous versions. On macOS, previously installed versions remain.

Sometimes it can be useful to have access to multiple versions of the software product. This can be done by copying the entire installation directory hierarchy to another location.

Use such copied installations by locally adjusting your `PATH` environment variable or by using an explicit path in the command line.

To remove a copied product, simply delete it. Do not run the `uninstall` script, as it will affect the normally installed copy.

Note: The software product is licensed under terms that govern copying and how it can be used.

CHAPTER 3

Tutorial

This chapter shows how to build and run a simple "Hello World" application on different targets.

The command line examples offer a clear understanding of the process. Tools can also be used with an IDE, but setup varies.

Beyond the Calypsi C compiler tool chain, you will need Make to control builds and Git to obtain projects. An internet connection is also required to download example projects.

3.1 Simulator

The `db68k` contains a simulator mode which is enabled by default. It mimics the 68000 architecture and allows the application to run on your host machine.

From a terminal or command line, copy the example project from the internet using:

```
$ git clone https://github.com/hth313/Calypsi-m68k-hello-world.git
```

Change to the newly cloned project directory:

```
$ cd Calypsi-m68k-hello-world
```

Build the project with:

```
$ make
cc68k --core=68000 --code-model=large --data-model=small --debug --list-file=obj/main-debug.lst -o obj/
↪main-debug.o src/main.c
ln68k --debug -o hello.elf obj/main-debug.o module/Calypsi-m68k-Foenix/linker-files/a2560u-simplified.
↪scm clib-68000-lc-sd.a --list-file=hello-debug.lst --cross-reference --rtattr printf=reduced --semi-
↪hosted --target=Foenix --stack-size=2000 --sstack-size=800
```

This will create an output file `hello.elf`. To run the application you can load it into the debugger using:

```
$ db68k hello.elf
Calypsi debugger for 68000
(db68k)
```

By default, the debugger enters interactive command mode. Run the application using the `run` command:

```
(db68k) run
running
Hello World!
program exited normally
(db68k)
```

When you are done with the debugger, you can leave it using the `quit` command:

```
(db68k) quit
$
```

You can also run the program directly from the command prompt, without entering interactive mode. You need to provide the debugger with additional command-line options to run and terminate automatically:

```
$ db68k hello.elf -e run --terminate-on-program-exit
Calypsi debugger for 68000
run
running
Hello World!
program exited normally
$
```

This tells the debugger to execute the command `run` and that termination of the application also means termination of the debugger.

3.2 Amiga

This section describes how to build a "Hello World" application for the Amiga.

For the Amiga we need to generate Hunk output and the output can either be executed on an emulator such as FS-UAE or WinUAE.

First you need to Git clone the Amiga Hello World example, you can do so using:

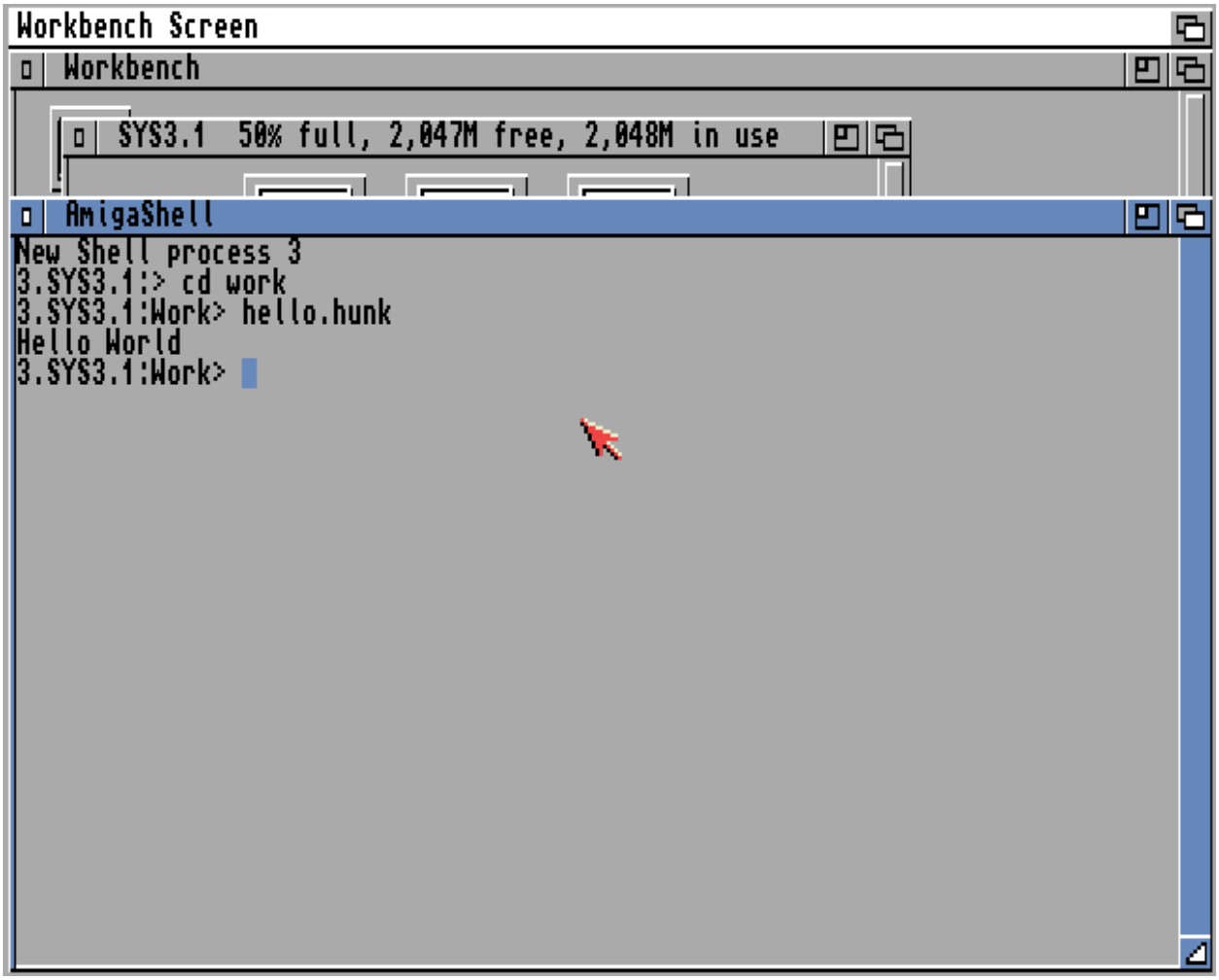
```
$ git clone https://github.com/hth313/Calypsi-Amiga-hello-world
```

Change directory to the just cloned project and build the project:

```
$ cd Calypsi-Amiga-hello-world
$ make
```

This will generate an `hello.hunk` output file. You need to copy this file to your emulator. Consult the documentation of the emulator you are using. For the FS-UAE it will probably use a directory on the host computer as the Amiga hard drive, copy `hello.hunk` to a suitable location on it and start the emulator. You may want to rename the file to be without `.hunk` while doing the copy.

Start the emulator, open a shell and run the program. It should print "Hello World!" and then exit:



3.3 A2560 Foenix

This section shows how to build the "Hello World" application for a A2560 Foenix and run it on hardware.

This means you need to have access the actual hardware, but it is possible to use an A2560 emulator as well, such as Morfe.

The the Calypsi C compiler tool chain generates ELF/DWARF by default, but you may prefer to use the native PGZ format instead. It is also possible to load an Intel-hex file over the debug port on the A2560. The provided `Makefile` already has a target to build a PGZ executable.

If you have not previously cloned the project, you can do so using:

```
$ git clone https://github.com/hth313/Calypsi-m68k-hello-world.git
```

Change directory to the just cloned project:

```
$ cd Calypsi-m68k-hello-world
```

When you are standing in the project folder, you can build the application for the A2560 Foenix using:

```
$ make hello.pgz
```

This will result in a program file `hello.pgz`. In order to run it you need to put it on the removable SD card storage media you have with your A2560 Foenix.

Once you have the hardware or emulator set up you can load and run the `hello.pgz` by typing the filename at the prompt.

Downloading an executable to a real A2560 can also be done using the USB debug port and the C256 Manager tool. In this case you will need to generate an S-record output file.

3.4 The Morfe emulator

It is also possible to download the program into Morfe, the emulator. Currently this need to be done by a Intel-hex file which can be generated using:

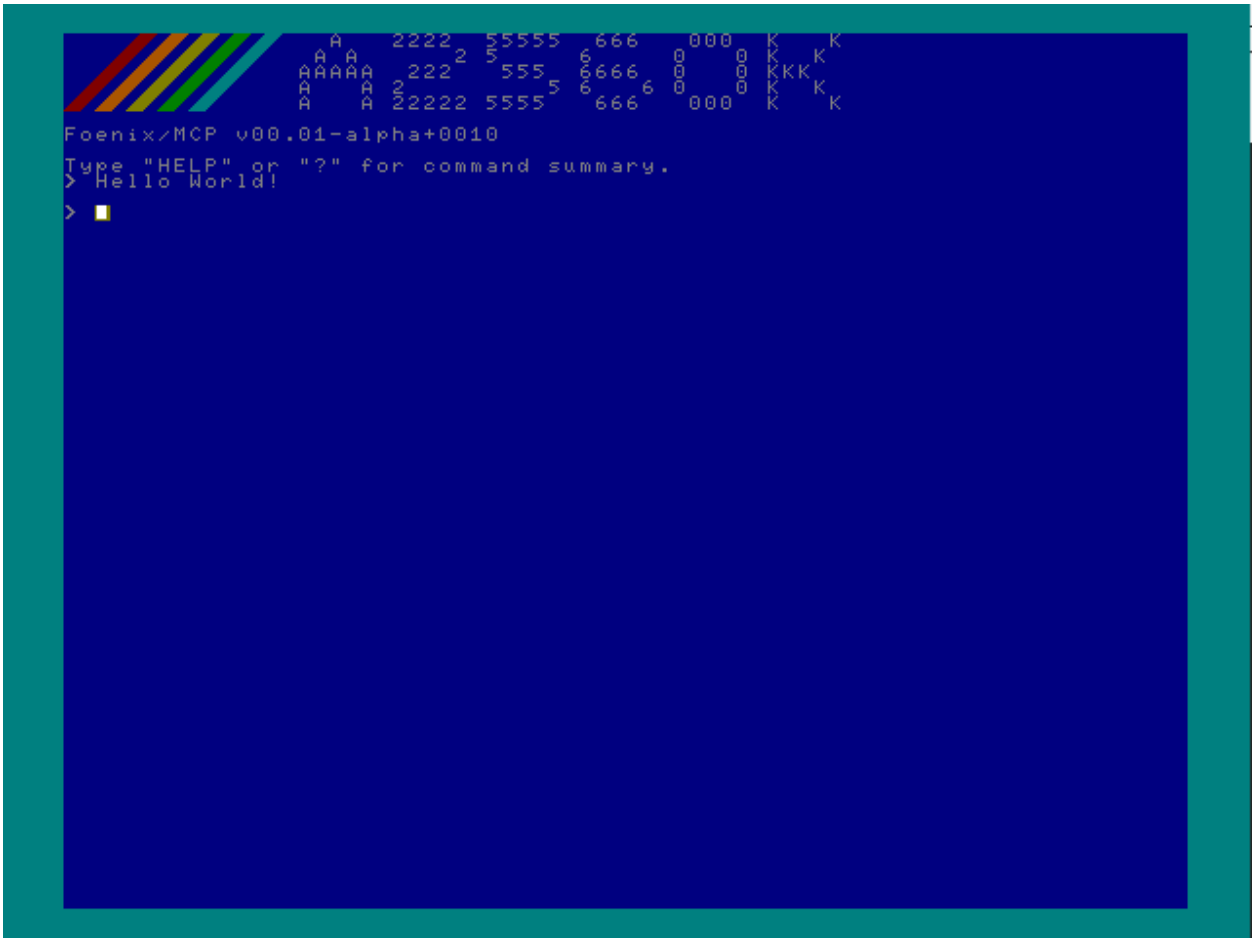
```
$ make hello.hex
```

The Morfe emulator can be found at https://github.com/aniou/morfe and need to be built according to instructions found there. Morfe can be launched emulating the A2560 K:

```
$ ./morfe-m68k conf/a2560k.ini
```

Once started, hit function key 9 to enter its TUI environment. Here you can load the Intel-hex file using `load hex hello.hex`. Then you may need to set the `PC` register manually using `set pc`. In this case you may need to look up the address in the generated `hello-Foenix.lst` file and look for the address of `__program_start` in it.

Once entered, you can press `quit` to resume the emulator at the start of the program. The result should be:


Foenix/MCP v00.01-alpha+0010
Type "HELP" or "?" for command summary.
> Hello World!
>

CHAPTER 4

Getting started

This chapter provides an overview of using the compiler and its related tools.

4.1 C language

C is a widely used programming language. It is well-suited for hardware-level programming and also functions as a powerful, generic high-level language. It offers powerful abstractions over target machines, enabling efficient application development and precise control.

This implementation uses the ISO/IEC 9899:1999 standard, commonly known as C99. In this guide, it is referred to as Standard C.

4.1.1 Cross compiler

The compiler is a cross-compiler, running on a modern workstation but producing applications for more constrained target machines.

4.1.2 Supported devices

This Calypsi C compiler tool chain supports the Motorola 68000, 68010, 68020, 68030, 68040 and 68060. Additionally, some support is provided for the APOLLO 68080 core.

4.2 File extensions

The following table shows the file extensions normally used with the Calypsi C compiler tool chain.

Table 4.1: File extensions

Extension	Purpose
`.c`	C source
`.h`	C header source
`.s`	Assembler source
`.o`	ELF/DWARF object file
`.a`	Library (collection of object files)
`.lst`	List file
`.scm`	Linker rules
`.elf`	ELF/DWARF output (executable file)
`.hex`	Intel-hex output
`.srec`	Motorola S-record output
`.s19`	Motorola S-record output, 16-bit address records
`.s28`	Motorola S-record output, 24-bit address records
`.s37`	Motorola S-record output, 32-bit address records
`.raw`	Raw output
`.pgz`	Foenix binary format
`.prg`	TOS binary format
`.hunk`	Amiga Hunk binary format

4.3 Building applications

Applications can be built from source files and libraries. Source files, written in C or assembly language, are compiled into object files: C source files use `cc68k`, and assembly source files use `as68k`.

A library is a collection of object files produced by the `nlib` tool, combining them with an index into a single file. The C runtime library is provided as an example; third-party libraries are also supported.

The `ln68k` takes object files, libraries, and placement rules as input to construct the executable application.

4.3.1 Compiler

The compiler command-line interface processes a single source file to produce an object file:

```
$ cc68k source.c
```

The object file produced will have the same base name as the input but with a `.o` file extension.

Note: You typically need command-line options to select the CPU core, runtime models, and other settings.

4.3.2 Assembler

C projects do not require knowledge of assembly language. C simplifies programming and enables portability across architectures.

However, for highly specific target code, deep-level control, or critical routines, the assembler is indispensable.

The assembler command-line interface is similar to the compiler, with the main difference being the file extension:

```
$ as68k source.s
```

The produced object file has the same `.o` extension as the compiler.

Note: You may need to provide a `--core` option for the assembler to recognize the target exact machine instructions.

4.3.3 Linker

The linker combines object files and libraries to create an executable application.

A rules file is required by the linker to describe the memory system, including placement rules for code and data. Stack and heap sizes can also be specified in this file.

You can run the linker from the command line as:

```
$ ln68k myfile1.o myfile2.o rules.scm
```

This simplified command produces `aout.elf`, an ELF binary.

There are many ways to tailor the output:

- Using hex output, in either Intel HEX or Motorola S-record file format
- RAW output, which is just plain binary output of a single memory area
- Foenix program files, which are segmented binary files with start address
- Amiga Hunk relocatable format
- TOS which is the binary format used by the Atari ST platform
- The `--debug` option includes DWARF debugging information in the ELF executable.

The linker can produce a list file with cross-reference information, showing memory usage, placement, and why certain library contents were included.

4.4 Configuring

Tune compiler code generation using various command-line options.

The most basic settings to consider are:

- The CPU core (`--core`) in use. This controls the exact instruction set.
- The data model, which affects how data are placed and accessed in memory.
- The size of the `double` floating point type.

- Optimization settings.

4.4.1 Core

The Calypsi C compiler tool chain currently supports the 68000, 68020, 68030, 68040, 68060 and the APOLLO 68080 cores.

4.4.2 Code model

The code model controls which instruction is used to make a function call.

4.4.3 Data model

The data model controls where global data is located by default.

The available data models are Small, Large and Far-only. If not specified the compiler uses the Small data model by default.

All data models are capable of addressing the entire 32 bits address space. The difference between the data models is in whether register `A4` is used as a base pointer.

Small

In the Small data model register `A4` is used as a base pointer to a 64K area where static and global variables are allocated. This typically saves two bytes for each each instruction that access such variable directly.

You can still allocate larger objects outside this area by using the Far attribute.

Large

In the Large data model variables can be allocated anywhere in the address space and the compiler will use 32 bits absolute addressing which typically results in large code compared to the Small data model.

The base register `A4` is still regarded as a base pointer, but is left unused. This is intended to allow writing library code that can be called from other applications that use `A4` as its base register.

Far-Only

The Far-Only data model is similar to the Large data model, but it treats register `A4` as a general purpose register that can be used as any other address register.

Table 4.2: Data models

Data model	Default pointer size	Default size limit
Small	32 bits	64K bytes
Large	32 bits	4G bytes
Far-Only	32 bits	4G bytes

4.4.4 Size of double

The `double` floating-point type uses IEEE 754 format and can be set to either 32 or 64 bits using `--32bit-doubles` or `--64bit-doubles`. It defaults to 32 bits if not specified.

The `float` type is always 32 bits, and the `long double` type is always 64 bits.

4.4.5 Optimization

Select the optimization level using the `-O` command-line option, which accepts a numeric argument of `1` or `2`.

The compiler applies a number of techniques to reduce the width of expressions, select efficient code sequences and dead code elimination, regardless of optimization settings.

The `-O` option enables further optimization to reduce the code memory footprint and typically increase execution speed.

4.5 Low level control

This section provides a brief overview of controlling access to specific memory and built-in functions, also known as intrinsics.

4.5.1 Extended keywords

Memory attribute keywords are provided for `__near` and, `__far` memory. The `__near` keyword is for register `A4` base relative addressing. The `__far` keyword is full 32 bits addressing which can be useful for larger static data objects when using the Small data model.

You can specify an attribute such as Near in two ways, either as a `__near` or `__attribute__((near))`.

Function attributes includes `__saveds` and `__farfunc`. The `__saveds` attribute is useful in the Small data model. It makes a function callable from outside your application. Such call context will not share the same register `A4` base address as your application. The compiler will generate code to preserve the old register `A4` value and initialize it with the base address context used in your application.

4.5.2 Intrinsics

Intrinsics functions appear as ordinary calls but are special compiler constructs, emitting specific instruction sequences. To enable them, include the `calypsi/intrinsics68000.h` header file.

4.5.3 Assembly code

You can write functions in assembly language by following the C calling convention. These functions can be called from C in the same way as any C function.

CHAPTER 5

Data storage

This chapter discusses controlling data placement and its impact on application efficiency.

The 68000 is a 32 bits architecture that can address a linear 4GB address range which is shared by code and data.

As there is a good amount of storage registers and your statically allocated data is probably not more than 64K, you can normally use the Small data model which reserves one register as a base address for addressing static data. This typically saves two bytes of program space for every instruction that access static data.

5.1 Ways to store data

Data can be allocated as auto, static, or on the heap. Variable scope and the need for allocation during runtime determine placement.

As a general rule, use auto-allocated variables whenever possible. This offers the compiler maximum flexibility for resource allocation closest to the core, which typically results in the most efficient data access.

5.1.1 Auto variables

Auto variables include function parameters and local variables not defined with `static`. The compiler attempts to allocate these in processor registers; otherwise, the stack is used.

Auto variables are allocated only when used. Their registers can be reused for other auto variables or temporary data once no longer needed. This reuse also applies to stack locations.

Auto variables can have multiple live ranges; a variable with the same name might be used in distinct parts of a function. Internally, these are treated as different variables and may be allocated to various locations, potentially not existing between live ranges.

All auto variables associated with a function are deallocated upon function exit.

Note: If you take the address of an auto variable, you can pass its pointer to other functions, which is useful for temporary storage. However, avoid using such pointers outside their scope, as auto variables are deallocated on function exit. Be aware that taking an address allocates the variable on the stack, potentially increasing access cost compared to other auto variables.

5.1.2 Static variables

Global, module, or function static variables are allocated in global memory, occupying space for the application's duration.

Their visibility varies. A global variable is universally visible but requires `extern` for use. A module static variable, declared with `static` at file scope, is visible within one compilation unit. A static local variable within a function's scope is only visible there, retaining its value across function calls. Use `static` to differentiate it from an auto variable.

5.1.3 Dynamically allocated

A dynamically allocated variable is retrieved from a heap using the malloc function. This is useful when the required data size is unknown at program startup.

Note: Dynamically allocated variables are a potential problem in memory constrained systems if the program is left running for a long time due to heap fragmentation.

5.2 Address spaces

The compiler provides multiple address spaces, which are addressable memory areas with specific properties:

- Address width for pointers
- Width of the associated integral index type
- Different instruction sequences for accessing various address spaces
- An extension keyword or type attribute name
- Section names tied to the address space for linking control

Address space attributes are always active in the compiler.

The Calypsi C compiler tool chain for the 68000 provides two address spaces, near and far.

All pointers to data memory are 32 bits wide and occupy 4 bytes of memory when stored in memory. The keywords and address space attributes are intended to describe a storage location to help placement in appropriate sections.

5.2.1 Near address space

The near address space is a memory area reachable from a base address held in register `A4`. This can be used in either the Small or Large data models, but it is not available in the Far-only data model.

You can have a maximum of 64K data in the near address space. If you run out of space you may select a couple of larger data objects and move them to the far address space (see below). 64K can be considered quite a lot of memory for storing reasonable small static data objects. Stack and heap allocated objects are allocated outside this 64K memory range.

5.2.2 Far address space

The far address space makes use of the full 32 bit address range. The amount of static data objects can be up to 4GB. Direct access to the far address space costs an additional two extra bytes for each instruction compared to using the near address space.

Note: While doing a direct access to a far object costs two extra bytes, there is no difference when using a pointer to the near or the far address area, they have the same cost.

5.2.3 Summary

The following table summarizes the available address spaces.

Table 5.1: Address spaces

Memory type	Keyword	Address range	Pointer size	Index type
Near	`__near`	`-0x8000` to `0x7fff`	32 bits	`int32_t`
Far	`__far`	`0x00000000` to `0xffffffff`	32 bits	`int32_t`

Note: The Near memory type allocates data in a 64K bytes large area pointed to by a base pointer held in register `A4`. Pointers and address arithmetics are performed using 32 bit operations.

5.2.4 Syntax

An address space attribute keyword such as `__far` is a type qualifier. Syntactically it works the same as other C language defined type qualifier, e.g. `const` and `volatile`.

The following declaration defines four variables in the Far memory address space:

```
__far int a, b;
int __far c, d;
```

The `__far` type qualifier applies to the closest type, `int` in this example. In C, the order between type qualifiers and types does not matter; they convey the same meaning.

5.2.5 Pointers

A pointer in C points to something in memory. Both the pointer itself and what it points to have types. As an example, the type `char *` is a pointer to a `char`.

Pointer types are easier to understand if you read them from right to left. The `*` is a pointer, so `char *` reads from right to left as "pointer to char". This order of reading is especially useful when you mix in type qualifiers in pointer types, as it makes it a lot easier to read and understand what the type means.

```
int __attribute__((far)) * p1;
long * __attribute__((far)) p2;
```

Here, `p1` is a pointer stored in default memory that points to an `int` in Far memory memory. `p2` is a pointer stored in Far memory memory that points to a `long` in default memory.

5.2.6 Structures

You can place a structure in a specified address space. This means that all its members are in that address space. You cannot override individual structure members using an address space keyword. It is however possible to have members of the structure that point to a different address space.

```
struct tag {
  int __far * p;
  int value;
};

struct tag __far myTag;
```

This is however not allowed:

```
struct tag {
  int * __far p;     /* incorrect */
  int __far value;   /* incorrect */
};
```

CHAPTER 6

Placing code and data

When compiling for modern computers, code and data placement often requires minimal attention, as tools handle it effectively for same-machine targets.

Cross-compilation is more complex, requiring knowledge of the target system. Embedded systems are even more challenging, with widely varying memory systems necessitating precise control over code and data placement.

Effective embedded system tools provide control over code and data placement. This is achieved using building blocks that offer placement flexibility and control. The compiler assists by providing abstractions through type qualifiers, initializers, and runtime model settings, which simplify configuring the memory system.

6.1 Sections

A section is a unit of code or data. The compiler will place code and data into different sections which are saved as separate sections in the object file.

The compiler places data objects and functions into suitable sections following specific rules. For data objects, it considers if the object is `const`, its address space, and explicit initializers.

Table 6.1: Sections

Section name	Type	Memory kind	Description
code	text	ROM	Executable code
nearcode	text	ROM	Executable near code
znear	bss	RAM	bss (zero initialized) near data
near	data	RAM	Initialized near data
zfar	bss	RAM	bss (zero initialized) far data
far	data	RAM	Initialized far data
cfar	rodata	ROM	Constant far data
switch	rodata	ROM	Switch tables
inear	rodata	ROM	Near data initializers
ifar	rodata	ROM	Far data initializers
data_init_table	rodata	ROM	Data initializer table
reset	text	ROM	Reset vector, when used
heap	noinit	RAM	Heap memory, for `malloc()`
stack	noinit	RAM	User stack
sstack	noinit	RAM	Supervisor stack

The sections `inear`, `ifar` and `data_init_table` in the table above are linker generated.

Note: It is assumed that there are no ROM or flash in either the Near area. Constants placed in either of those are placed in a normal writable section and the `const` attribute only affects the type of the object.

Note: The table assigns sections to ROM and RAM for ROM-based applications that start on power-up. For hosted systems loading from storage to RAM, this distinction is not applicable. Always consider ROM-marked sections as read-only.

Section types are derived from the UNIX world; see Section kinds in the assembler reference for details.

Note: When generating a ROM-executable application, the linker separates an initialized data section into two. Initializers go into a new `i`-prefixed section (placed in ROM); the data section is placed in RAM. The C runtime startup module copies initializers from ROM to RAM before giving control to the `main()` function.

6.1.1 Linking process

The linker reads object files with sections and a placement rules file. This file describes available memory areas and their permitted sections, optionally with additional placement constraints.

Memories containing actual value bits (program code or initialized data) are emitted in the executable.

An executable can serve as input for ROM or FLASH programmers, be loaded by the target machine, or be used by a debugger that connects to or simulates the target. Various executable formats are available to suit different execution environments.

Note: Although code and data are entirely separated into different sections, placing these sections together in the same memory may be desirable. This can be achieved at the link stage.

6.1.2 Placing sections in memory

The linker rules file uses file extension `.scm`, which is actually a source file in the Scheme language. However, you do not need to learn Scheme in order to use it, as it just a simple description of available memories and their associated properties. Sections are then bound to the memories. You can also define the size of the stack and heap in this file.

A simple example of such linker rules file is:

```
(define memories
  '((memory flash (address (#x100000 . #x1fffff)) (type ROM))
    (memory RAM   (address (#x200000 . #x23ffff)) (type RAM))
    (memory Vector (address (#x0000 . #x03ff))
            (section (reset #x0000)))
    ))
```

The Scheme interpreter evaluates the rules file, which defines a single global `memories` object. This object lists `memory` and block objects. Sections can be bound to memories either by specifying the memory type or by listing the sections within each memory.

The example simplifies memory system description. The linker deduces where compiler and linker-generated sections are placed; this requires specifying the memory type. To prevent automatic section placement in a memory, omit its type specification.

Note: You can specify a memory type and manually place custom sections. This is useful for sections requiring specific placement.

Note: Since the linker rules file is a Scheme program, it can theoretically construct the `memories` object via functions or macros. After evaluation, the Scheme interpreter is expected to leave a `memories` object in its global environment. However, a simple file like the example is usually sufficient.

6.2 Data initialization

A startup routine is performed before the `main()` function is called. This startup routine takes care of preparing the execution environment for the C program by initializing the stack pointer, initialize global variables and prepare system resources such as file streams and the heap, if used by the application.

6.2.1 Static data

Static data that needs to be initialized comes in two forms, zero initialized and those that have non-zero initializers. Zero initialization is done by filling such memory ranges with zero. Explicit initialization is done by copying an initializer section in ROM to its corresponding RAM area.

This process is table driven using a section named `data_init_table` which should be placed in the same memory as the `!cdata` section, which holds constants.

Note: In a hosted environment, the linker provides the `--hosted` command-line option. This assumes the program image loads directly into RAM from external media. Variable initialization is done in-place via the load action; therefore, separate initializers are not required.

The `--target` option also has the effect of enabling running in a hosted environment.

6.3 Stack

The stack stores function return addresses, saved registers, and local variables. It tracks the current execution state, enabling reentrant and recursive functions.

6.3.1 Stack size considerations

The stack size is defined in the linker rules file. The stack size should be set low as possible, but not any lower. If given too much space you are wasting RAM space not being used for anything. However, if set too small it will cause corruption of other data.

Exact stack use can be hard to predict and it is better to err on the side of some safety. Also during development of your application you may want to use a bit oversized stack if possible, to allow you to focus on other issues. In the end, you are most likely constrained by some fixed amount of RAM and need to balance how much you can use for the stack with the optional heap and statically allocated data.

CHAPTER 7

C language

This chapter details the C language implementation, supporting the ISO/IEC 9899:1999 standard (C99).

7.1 Overview

For those accustomed to older C versions, C99 introduces several useful features:

- Declarations and statements can be mixed in the same block scope
- A declaration can be placed in the initialization expression of a `for` loop
- The `bool` datatype, available as `_Bool` or as `bool` after the `stdbool.h` header file has been included
- The `long long` data type
- C++ style comments
- Use of incomplete array at end of a structure
- Variable length arrays
- Initialization of compound objects by the use of designated initializers improve readability

7.2 Extensions

Language extensions are always enabled, encompassing various qualifier keywords, intrinsic functions, and additional library features.

CHAPTER 8

Efficient coding

The Calypsi C compiler tool chain is designed for coding in memory-constrained environments, typically embedded systems. This chapter explores proper data type usage and considerations for both smaller and larger targets.

Even if your current coding targets a specific system, the code may evolve and migrate to different system sizes. This chapter discusses trade-offs between such environments.

8.1 Data types

Data type selection significantly impacts overall program size and execution speed. Here are some rules of thumb:

- Generally, use the smallest possible types.
- Floating-point operations are typically inefficient, consuming space and often executing slowly.
- Use variables with the narrowest possible scope.
- Avoid taking a variable's address. If necessary, limit its use as much as possible. The compiler cannot allocate such variables in registers (registers have no addresses), which prevents the optimizer from fully understanding how function calls affect the variable.

8.1.1 Integer types

While smallest types are recommended, for larger targets, `int` and `unsigned int` can be more efficient, often being the sweet spot.

The `stdint.h` header provides integer types useful for selecting appropriate sizes, optimizing efficiency, and ensuring portability.

A type like `uint_fast16_t` defines an unsigned integer of at least 16 bits, optimized for speed. An 8-bit target will use a 16-bit type, while a 32-bit target might choose a 32-bit type if its instruction set handles 32-bit operations more efficiently than 16-bit operations.

8.1.2 Floating point types

Floating-point operations are often implemented using assembly or C library routines, which consume code space and incur execution overhead.

Alternatively, use fixed-point integers (Q notation). For instance, a Q24.8 number has 24 integer bits and 8 fractional bits, allowing manipulation with standard integer operations. This requires scaling values, which is straightforward.

8.1.3 Implicit conversions

C's conversion rules aim to convert integers to `int` and `float` to `double`. This can introduce extra code. If this is a concern, you must understand these rules or examine the compiler's generated code.

Note: While studying generated assembly code may seem daunting, a basic understanding of its mechanics can be highly beneficial. The compiler can generate a list file (`-l` or `--list-file`) showing C code intermixed with the resulting assembly.

8.2 Use prototypes

Functions can be declared using two styles. Traditional C, which lacks prototypes, is not recommended as it hinders compiler error detection. Prototypes offer safer, more efficient code generation by avoiding certain type promotion rules that introduce extra code. Traditional style is supported only for compatibility with older code. In this style, a declaration does not specify parameters:

```
/* Traditional style */
int foo();   /* declaration */

int foo(c, i)   /* definition */
  char c;
  int i;
{
  return i - c;
}
```

In this case, the function takes a `char` parameter, which is promoted to `int` during the function call, introducing additional code.

Using prototypes with Standard C you specify parameters in the declaration:

```
/* Standard C prototype style */
int foo(char c, int i);   /* declaration */

int foo(char c, int i)   /* definition */
{
  return i - c;
}
```

In this case the `char` parameter is not promoted to `int` and is passed as a `char` type.

8.3 Use tables

Consider the trade-off between calculating a value and using a table for knowledge or operation encoding. For example, a sine operation in Q (fixed-point) notation can be approximated with a table lookup instead of calculation, making it very fast and compact.

CHAPTER 9

Target specifics

This chapter details the Amiga and A2560 targets.

9.1 Amiga

The installation provides the Amiga NDK (Native Development Kit) and an Amiga extension library to supplement the C library. The NDK supplied is version 3.2.

If you want to use a different Amiga NDK, search for "ndk" at Aminet[1].

The extension library project is available at Calypsi Amiga[2].

Using the `--target=amiga` command-line option enables Amiga headers and knowledge of its system libraries. This provides direct access to Amiga system headers and ensures proper parameter passing to Amiga library entry points.

Note: If you use a different Amiga NDK, treat its headers as system headers using `#include <header.h>`. Specify the path with `--include-system` to place them before compiler-supplied header files.

---

[1] http://aminet.net
[2] https://github.com/hth313/Calypsi-Amiga

9.1.1 Auto open libraries

Amiga libraries must be opened before use. The compile-time runtime automatically opens certain libraries when you make function calls to them. This includes `dos.library`, `intuition.library`, and `graphics.library`. The `exec.library` does not require opening, as it is always present at address `4` on the Amiga.

Note: No specific version is requested when opening such libraries. If a specific version is required, open the library manually and handle any failures.

9.1.2 Amiga C runtime

The following cleanups occur automatically upon a normal application `exit()`:

1. All memory allocated using `malloc()` is released.
2. All opened files are flushed and closed.
3. Automatically opened libraries are closed. This includes `dos`, `graphics` and `intuition` libraries.

Errors from Amiga DOS calls, made on behalf of the C library, are translated to C-style error codes and stored in the `errno` variable.

9.2 Foenix A2560

The installation provides board support for the Foenix A2560[3].

Using the command-line option `--target=a2560k` or `--target=a2560u` makes specific header files available for access:

```
#include <mcp.h>
#include <foenix.h>
```

A2560 linker rule files are included in the installation directory. They follow an `a2560` prefix naming convention (e.g., `a2560u+.scm`). You can refer to these files by name without specifying the full path. The linker first checks the current directory, then the installation directory if not found.

To modify a `.scm` file, copy it to your project and edit as required.

When linking, specify the same `--target=a2560k` or `--target=a2560u` option. This enables access to additional target-specific board support libraries, which are named with an `sdk` prefix instead of `clib`.

[3] https://github.com/hth313/Calypsi-m68k-Foenix

CHAPTER 10

Running the compiler

The compiler is run from the command line, or from an IDE that interfaces with the compiler using the command line.

10.1 Basic invocation

The compiler is invoked in the following way:

```
$ cc68k [options] sourcefile [options]
```

As an example, to compile a source file with debugging information and a list file, you can use:

```
$ cc68k --debug source.c -l
```

Command line options are optional arguments that tune the behavior of the compiler. They always start with a dash character. There are two variants, single letter options (`-l` to instruct the compiler to create a list file) starts with a single dash. The other variant is long descriptive options that starts with two dashes.

Some options require arguments. These follow the option, separated by a space or an equals sign (`=`). For single-argument options, the separator is optional:

```
$ cc68k -Iinclude source.c -D VERBOSE=2 --list-file=tiny-source.lst
```

In this case the `-I` option adds `include` as a directory to scan for header files. The symbol `VERBOSE` is defined in the preprocessor with value 2. A list file with a specific name is also produced.

The order in which the options appear normally do matter, except for the `-I` option that adds directories to search for header files. Such directories are searched in the order in which they appear on the command line.

To display the version of the compiler, use `--version`:

```
$ cc68k --version
Calypsi ISO C compiler for Motorola 68000 version 5.18
```

Command line options are described in Command line options.

10.2 Include search path

Header files are included by surrounding the filename with either double quotes or angled brackets:

```
#include "myheader.h"
#include <stdio.h>
```

Angled brackets are typically for system header files, while double quotes are for application-specific header files.

The search order for include files is as follows:

1. Relative to compilation directory (double quote include only)
2. In directories specified in the `-I` option in the order they appear on the command line
3. The system directory, which is the installation directory

Header files specified in angle brackets are not searched relative to the compilation directory. Otherwise, they behave identically. The system header file directory is located within the installation directory.

Note: More precisely, the system header file directory is relative to the cc68k executable. If an installation is moved, it will still find the correct system header file directory.

10.3 Compiler output

The compiler outputs one or two files, an object file that can be linked with other object files and libraries by `ln68k` to produce an executable file, and optionally a list file.

10.3.1 Object file

The object file is in ELF format, optionally including DWARF debugging information if `--debug` (or `-g`) is specified.

It contains:

- A symbol table
- Relocatable sections for code and data
- Relocations, allowing linker to modify address values after section placement
- DWARF-formatted source-level debugging information (if `--debug` was specified)
- Vendor-specific data, including runtime attributes and additional the Calypsi C compiler tool chain-specific information not covered by ELF/DWARF, used by the linker and debugger, with section types `0x8000000a` and `0x8000000b`.

Note: DWARF version 5 is used. The implementation covers the needs of the Calypsi C compiler tool chain and includes vendor extensions.

10.3.2 List file

The list file is a text file meant to be shown with a fixed width font. It contains a header that shows compiler, version, time when it was created and the command line used. The C source file follows intermixed with generated assembly code. Finally there is a summary of the code and data sizes.

The following simple summation function:

```
int sum(int a, int b, int c) {
  if (a < b) {
    return a + c;
  } else {
    return b + c;
  }
}
```

Compiling with the `-l` option produces a list file:

```
###############################################################################
#                                                                             #
# Calypsi ISO C compiler for Motorola 68000                    version 5.18 #
#                                                          15/Jul/2026  16:44:46 #
# Command line: sum.c -l                                                      #
#                                                                             #
###############################################################################

    \ 00000000                      .rtmodel version,"1"
    \ 00000000                      .rtmodel nearDataBase,"A4"
    \ 00000000                      .rtmodel core,"68000"
    \ 00000000                      .rtmodel codeModel,"large"
    \ 00000000                      .rtmodel target,"none-specified"
0001                    int sum(int a, int b, int c) {
    \ 00000000                      .section code,text
    \ 00000000                      .public sum
    \ 00000000                      .align  2
    \ 00000000 2f02     sum:        move.l  d2,-(sp)
    \ 00000002 242f0008             move.l  (8,sp),d2
0002                      if (a < b) {
    \ 00000006 b280                 cmp.l   d0,d1
    \ 00000008 6f06                 ble.s   `?L4`
0003                        return a + c;
    \ 0000000a d480                 add.l   d0,d2
    \ 0000000c 2002                 move.l  d2,d0
    \ 0000000e 6004                 bra.s   `?L3`
    \ 00000010          `?L4`:
0004                      } else {
0005                        return b + c;
    \ 00000010 d481                 add.l   d1,d2
    \ 00000012 2002                 move.l  d2,d0
    \ 00000014          `?L3`:
0006                      }
0007                    }
    \ 00000014 241f                 move.l  (sp)+,d2
    \ 00000016 4e75                 rts

#########################
#                       #
# Memory sizes (decimal) #
```

(continues on next page)

(continued from previous page)

```
#                         #
###########################

Executable  (Text): 24 bytes
```

10.3.3 Assembly source

The compiler can generate assembly source output, which is similar to a list file, except that this output file is actual source code that can be given to the assembler. This can be used to generate a starting point for an assembly source file. By providing suitable declarations and simplified C functions you can study what the compiler does and build upon it.

10.4 Diagnostics

Compiler diagnostic messages identify problems that either must or should be addressed.

10.4.1 Errors

An error identifies a problem in the code that prevents the compiler from producing a valid output file.

10.4.2 Warnings

A warning identifies a potential problem in the code that you probably want to take a look at and possibly rectify. The compiler is still able to produce a valid output file.

10.4.3 Internal errors

An internal error indicates an unexpected problem within the compiler product.

10.4.4 Controlling warnings

Diagnostic can be controlled using the `-W` option that takes a wide range of possible alternatives. It is beyond the scope of this guide to list them all. A complete reference can be found at https://clang.llvm.org/docs/DiagnosticsReference.html as the parser and diagnostics engine is the same as in the Clang project.

Consider the following code:

```
int test(int x)
{
  if (x = 6) {
    return 10;
  } else {
    return 0;
  }
}
```

In the code, an assignment is used in a comparison, a common C error. While legally testing a non-zero assignment expression, this often indicates an intended use of ==. The compiler cannot know your intent, but flags a potential problem. Compiling this code results in warnings:

```
w.c:3:10: warning: using the result of an assignment as a condition without parentheses [-Wparentheses]
   if (x = 6) {
       ~~^~~
w.c:3:10: note: place parentheses around the assignment to silence this warning
   if (x = 6) {
         ^
       (   )
w.c:3:10: note: use '==' to turn this assignment into an equality comparison
   if (x = 6) {
         ^
         ==
```

The warnings point out that there may be a problem in the code. The name of the warning that triggered is also given (`-Wparentheses`). Finally, it makes suggestions on how you can change the code.

If you think the warning is of no use to you and you do not want to see this particular warning again, you can add the option `-Wno-parentheses` to the command line to silence it.

Adding an extra pair of parentheses around the assignment is a common way to indicate an intentional assignment within a test expression:

```
int test(int x)
{
  if ((x = 6)) {
    return 10;
  } else {
    return 0;
  }
}
```

Finally, if the intention with the code was to use the equality operator, you should change it to use the correct operator:

```
int test(int x)
{
  if (x == 6) {
    return 10;
  } else {
    return 0;
  }
}
```

10.5 Command line options

This section covers the `cc68k` command-line options in detail.

10.5.1 Options overview

If you run the compiler from the command line without any arguments it will complain that the input source file is missing and then shows a short form help:

```
$ cc68k
Missing: FILE

Usage: cc68k [--version] [-o|--output-file OUTPUT-FILE] [-l]
             [--list-file LIST-FILE] [-I DIRECTORY] [-D IDENTIFIER]
             [-U IDENTIFIER] [-g|--debug] ([--32bit-doubles] |
             [--64bit-doubles]) ([--char-is-signed] | [--char-is-unsigned]) [-E]
             [--print-macro-definitions] [--align-functions ALIGNMENT]
             [-O LEVEL] ([--space] | [--speed]) [--no-cross-call]
             [--no-interprocedural-cross-jump] ([--no-inline] |
             [--always-inline]) [--only-marked-as-inline] [--strong-inline]
             [--inline-on-matching-custom-text-section] [--rtattr NAME=VALUE]
             [--weak-symbols] [--no-vector-sections] [-c]
             [--assembly-source ASSEMBLY-FILE] [-S] [--force-switch STRATEGY]
             [-W ARG] [--dependencies] [-M ARG] [--pedantic-errors]
             [--include-system DIRECTORY] [--include-system-after DIRECTORY]
             [--core CORE] [--target TARGET] [--code-model NAME]
             [--data-model NAME] [--fd DIRECTORY] [--amiga-ndk-root PATH]
             [--no-amiga-ndk] [--pascal-strings] FILE
  use 'cc68k --help' for detailed help
```

For more detailed help, use the `--help` option:

```
$ cc68k --help
Calypsi ISO C compiler for Motorola 68000 version 5.18

Usage: cc68k [--version] [-o|--output-file OUTPUT-FILE] [-l]
             [--list-file LIST-FILE] [-I DIRECTORY] [-D IDENTIFIER]
             [-U IDENTIFIER] [-g|--debug] ([--32bit-doubles] |
             [--64bit-doubles]) ([--char-is-signed] | [--char-is-unsigned]) [-E]
             [--print-macro-definitions] [--align-functions ALIGNMENT]
             [-O LEVEL] ([--space] | [--speed]) [--no-cross-call]
             [--no-interprocedural-cross-jump] ([--no-inline] |
             [--always-inline]) [--only-marked-as-inline] [--strong-inline]
             [--inline-on-matching-custom-text-section] [--rtattr NAME=VALUE]
             [--weak-symbols] [--no-vector-sections] [-c]
             [--assembly-source ASSEMBLY-FILE] [-S] [--force-switch STRATEGY]
             [-W ARG] [--dependencies] [-M ARG] [--pedantic-errors]
             [--include-system DIRECTORY] [--include-system-after DIRECTORY]
             [--core CORE] [--target TARGET] [--code-model NAME]
             [--data-model NAME] [--fd DIRECTORY] [--amiga-ndk-root PATH]
             [--no-amiga-ndk] [--pascal-strings] FILE
  use 'cc68k --help' for detailed help

Available options:
  --version               Display version number
  -o,--output-file OUTPUT-FILE
                          Name of output file
  -l                      Generate a list file, named by appending '.lst' to
                          input file
  --list-file LIST-FILE   Generate list file, using given name
  -I DIRECTORY            Include directory
  -D IDENTIFIER           Predefine a macro
```

(continues on next page)

(continued from previous page)

```
-U IDENTIFIER            #undef a predefined macro
-g,--debug               Produce debugging information
--32bit-doubles          Make the 'double' data type 32-bits (this is the
                         default)
--64bit-doubles          Make the 'double' data type 64-bits
--char-is-signed         Treat 'char' as a signed type
--char-is-unsigned       Treat 'char' as an unsigned type (this is the
                         default)
-E                       Stop after preprocessing
--print-macro-definitions
                         Print macro definitions in -E mode in addition to
                         normal output
--align-functions ALIGNMENT
                         Align the start of functions to given alignment
-O LEVEL                 Enable optimizer, -O0, -O1 or -O2
--space                  Optimize for space (this is the default, use together
                         with -O)
--speed                  Optimize for speed (use together with -O)
--no-cross-call          Disable cross call optimization
--no-interprocedural-cross-jump
                         Disable interprocedural cross jump optimization
--no-inline              Disable all function inlining
--always-inline          Always inline functions
--only-marked-as-inline  Only consider inlining functions marked as 'inline'
--strong-inline          Always (try to) inline functions marked as 'inline'
--inline-on-matching-custom-text-section
                         Only inline custom text section functions when the
                         section names match
--rtattr NAME=VALUE      Define a runtime attribute (identifier or quoted
                         string value accepted)
--weak-symbols           Make all public symbols entries weak
--no-vector-sections     Discard auto generated interrupt vector sections
-c                       Generate object file and do not try to link
--assembly-source ASSEMBLY-FILE
                         Generate an assembly source file, using given name
-S                       Generate an assembly source file, named by appending
                         '.s' to input file
--force-switch STRATEGY  Switch strategy, one of 'if-else', 'jump-table' or
                         'value-table'
-W ARG                   Warning control
--dependencies           Generate dependencies to stdout
-M ARG                   Dependency file control
--pedantic-errors        Error on language extensions
--include-system DIRECTORY
                         Add system include directory
--include-system-after DIRECTORY
                         Add system include directory after
--core CORE              Core, one of '68000', '68010', '68020', '68030',
                         '68040', '68060' or '68080' (defaults to '68000')
--target TARGET          Target system, one of 'Amiga', 'A2560U', 'A2560K' or
                         'TOS' (defaults to embedded/ROM use, if omitted)
--code-model NAME        Code model, one of 'small' or 'large' (defaults to
                         'large')
--data-model NAME        Data model, one of 'small', 'large' or 'far-only'
                         (defaults to 'small')
--fd DIRECTORY           Add directory to search path for Amiga library .fd
```

(continues on next page)

(continued from previous page)

```
                        files
  --amiga-ndk-root PATH   Use specified Amiga NDK (instead of default one)
  --no-amiga-ndk          Do not automatically include Amiga NDK (active when
                          --target=Amiga is specified)
  --pascal-strings        Generate Pascal strings on leading \p character
  -h,--help               Show this help text
```

10.5.2 Options in detail

`--version`

Displays the name and version of the compiler.

`--output-file, -o`

Specify the output object file name. If omitted, the output file is derived from the input file name (excluding path) with a `.o` extension, written to the current directory by default.

This option can also alter the output file name and provide a directory path. The specified directory must already exist.

`-l`

Generate a list file. The name used is the name of the input file (ignoring any directory path) with a `.lst` file extension. See also `--list-file`.

`--list-file`

Generate a list file. The name of the list file is given as argument to this option. See also `-l` to generate a list file based on the source filename.

`-I`

Add a directory to the current include search path. This option can be used multiple times on a command line. The order in which they appear specifies the search order between the directories.

The system include directory is always added last to the search order list.

`--include-system`

Add a directory to the current system include search path before the provided system include directories. This option can be used multiple times on a command line. The order in which they appear specifies the search order between the directories.

`--include-system-after`

Add a directory to the current system include search path after the provided system include directories. This option can be used multiple times on a command line. The order in which they appear specifies the search order between the directories.

`-D`

Define a macro. This takes an argument with the symbol name and optionally an assignment value `-Dsymbol[=value]`. If no value is given, the macro is given the value 1.

`-U`

Undefine a macro. This takes an argument with the symbol name to be undefined.

`--debug`

Generate DWARF symbolic debugging information. In order to get debugging information all the way to the debugger, the linker must also be given this option.

When debugging information is enabled the `__CALYPSI_DEBUG__` macro is also defined and set to 1.

`-g`

Synonym for `--debug`.

`-S`

Generate an assembly source file as output. The name used is the name of the input file (ignoring any directory path) with a `.s` file extension. No object file is generated. See also `--assembly-source`.

`--assembly-source`

Generate an assembly source file as output. The name of the output file is given as an argument to this option. See also `-S` to generate an assembly source file based on the C source filename. No object file is generated.

`--32bit-doubles`

The `double` floating point data type has 32 bits precision.

`--64bit-doubles`

The `double` floating point data type has 64 bits precision.

`--char-is-signed`

Makes the `char` data type signed. The default is unsigned.

`--char-is-unsigned`

Makes the `char` data type unsigned. This is also the default.

`-E`

Stop after running the preprocessor. The output from the preprocessor is written to standard output (`stdout`).

`--print-macro-definitions`

This is used together with the `-E` option to also write out all macro definitions.

`-O`

Runs the optimizer, expecting an argument of 0, 1, or 2. `-O0` (the default) performs no additional optimization passes but still produces good code. `-O1` enables additional passes to further improve code, and `-O2` enables all optimizer passes.

`--space`

Optimize for space, this is the default. Use `-O` to enable optimizations.

`--speed`

Optimize for speed. Use `-O` to enable optimizations. This option controls decisions about tradeoffs and will balance towards making the code run faster at the expense of additional code space. See also `--no-cross-call` below.

`--no-cross-call`

Disables the cross-call optimizer, normally enabled at `-O2`. The cross-call optimizer extracts common code sequences into small subroutines, which can significantly reduce code space, so it is also enabled by default with `--speed`.

`--no-inline`

Disable all function inlining which is normally enabled at `-O1`.

`--always-inline`

Enable function inlining regardless of optimization level setting.

`--only-marked-as-inline`

Only considers functions marked with the `inline` keyword for inlining. This disregards small functions and single-use static functions for inlining.

`--strong-inline`

Regard functions marked with the `inline` keyword as a strong hint that they should be inlined. The compiler may still decide not to do it.

`--rtattr NAME=VALUE`

This defines a runtime attribute which is written to the object file. This is used to specify runtime attribute value that can be used in the linker for checking object file consistency.

`--weak-symbols`

Makes all public symbols in the object file weak. This is normally not needed, but can provide a default library function implementation that is used if no replacement is provided.

`--no-vector-sections`

Do not generate vector table entries for interrupt functions. This is useful when writing interrupt functions that are going to be installed using some other mechanism.

`-c`

Generate an object file without linking. This is a no-operation, as the compiler always generates an object file. This option is provided due to its common use with many compilers.

`--force-switch`

Forces the compiler to use a specific strategy for generating switch tables. The compiler normally uses heuristics, but this option overrides them to force a particular variant.

Available variants are: series of "if-else" tests, a jump table, or a table with value and label pairs. "If-else" is often best for very small switches. A jump table is typically fastest for a decent number of close entries. Value and label pairs work well for larger tables with spread-out values, using a binary search lookup that scales fairly well.

In big O notation: "if-else" is O(n), a jump table is O(1), and value and label pairs are O(log(n)).

`-MD`

Write a dependency file that contains user and system headers.

`--dependencies`

Like `-MD`, but also implies `-E` and writes to `stdout` by default.

`-MF<file>`

Write dependency file output from `-MMD`, `-MD`, `-MM`, or `-M` to `<file>`.

`-MG`

Add missing headers to the dependency file.

`-MJ<arg>`

Write a compilation database entry for each input.

`-MM`

Like `-MMD`, but also implies `-E` and writes to `stdout` by default.

`-MMD`

Write a dependency file containing user headers.

`-MP`

Create phony target for each dependency (other than main file).

`-MQ<arg>`

Specify name of main file output to quote in dependency file.

`-MT<arg>`

Specify name of main file output in dependency file.

`-MV`

Use NMake/Jom format for the dependency file.

`-W`

Control warnings, see Controlling warnings.

`--pedantic-errors`

Generates errors when language extensions are used. The supported language includes relaxations to the strict C standard for increased flexibility; use this option to disable such extensions.

`--align-functions`

Aligns the start of all functions to the provided alignment.

`--core`

This selects the 68000 core to use. This can either be 68000 or 68010. This affects the available instructions. If not specified it defaults to 68000.

`--target`

This tells the compiler that the code is intended for a certain target platform. The supported platforms are Amiga and Foenix.

The Amiga support is currently work in progress and should be regarded as experimental. The Foenix target enables the code generator to make use of the hardware accelerated math unit, both for integer and floating point operations.

Specifying the Amiga or Foenix target options implies to the linker that it will use hosted behavior, see --hosted for more information.

Note: The Foenix math unit has several memory mapped registers that cannot be protected as an atomic block while using it. This means that an interrupt that uses the math unit will need to preserve the altered registers. More importantly, it also means that if a C written interrupt calls a function it will need to preserve all math registers which causes increased overhead in interrupts. To avoid this, you should avoid making function calls from interrupts, ensure called functions are inlined or compile the modules with such interrupts separately without the `--target=foenix` option.

`--code-model`

This options tells the compiler which code model to use. See Code model for more information.

`--data-model`

This options tells the compiler which data model to use. See Data model for more information.

CHAPTER 11

Data representation

This chapter describes the available data types in detail.

11.1 Basic types

All standard integer types are supported. The following table lists the built-in integer types and their ranges.

Table 11.1: Integer types

Type name	Size	Range
`bool`	8 bits	0 to 1
`char`	8 bits	0 to 255
`signed char`	8 bits	-128 to 127
`unsigned char`	8 bits	0 to 255
`short`	16 bits	-32768 to 32767
`signed short`	16 bits	-32768 to 32767
`unsigned short`	16 bits	0 to 65535
`int`	32 bits	-2^{31} to $2^{31}-1$
`signed int`	32 bits	-2^{31} to $2^{31}-1$
`unsigned int`	32 bits	0 to $2^{32}-1$
`long`	32 bits	-2^{31} to $2^{31}-1$
`signed long`	32 bits	-2^{31} to $2^{31}-1$
`unsigned long`	32 bits	0 to $2^{32}-1$
`signed long long`	64 bits	-2^{63} to $2^{63}-1$
`long long`	64 bits	-2^{63} to $2^{63}-1$
`unsigned long long`	64 bits	0 to $2^{64}-1$

11.1.1 bool

To use the `bool` data type, include `stdbool.h`, which also defines `true` and `false`. The boolean data type is also available as `_Bool` without requiring `stdbool.h`.

11.1.2 char

The `char` type is unsigned by default. To enable signed `char`, compile with the `--char-is-signed` option. Note that the supplied C library uses unsigned `char`.

Note: The `char` type differs from `short`, `int` and `long` in that it defaults to being unsigned in this compiler. Standard C allows it to be either signed or unsigned. The rationale for making `char` unsigned is that it is meant to represent a character code and in encoding standards like ASCII, ISO-8859-1 and Unicode, character values are unsigned entities.

If you intend to use 8 bits data types in expressions, you should consider using `int8_t` or `uint8_t` instead. They are defined in `stdint.h`.

11.1.3 wchar_t

The wide character type `wchar_t` is defined if you include `stddef.h`.

11.1.4 Bit fields

Bit fields are supported based on any integer type. A bit field value has the same type (and signedness) as the integer base type it is defined in and are subject to the usual conversion rules when used in expressions.

A bit field is allocated starting from the least significant available bitposition in its container. If there are not enough bits available in the container to represent the bit field, a new container is allocated.

Consider the following declaration:

```
struct bf {
  uint16_t  a:7;
  int16_t   b:5;
  uint32_t  c:24;
  uint8_t   d:4;
  uint8_t   e:2;
  int8_t    f:3;
};
```

The two bit fields `a` and `b` are allocated in the same 16 bits container. Bit field `c` gets a 32 bits container of its own. Finally, `d` and `e` will share the same 8 bits container while `f` is allocated in a separate 8 bits container as there is not room to store it together with `d` and `e`.

Bit fields should be used with care. They can be a convenient way to pack several small values into some structure when data space is limited. However, accessing bit fields is in general more costly in terms of produced code, compared to using normal integer types.

Note: Sometimes it can be tempting to try and map bit fields to hardware registers. This can work, but it makes the code more sensitive to using a particular compiler. It may also require some thinking to get it

right. The alternative way of accessing hardware registers in its intended access width and manually apply shift and mask operations is often more robust.

11.1.5 Floating-point types

Floating point values follows the IEEE 754 format and is supported in two different sizes.

Table 11.2: Floating-point types

Type name	Size	Approximate range (normal values)
`float`	32 bits	$\pm 1.18 \times 10^{-38}$ to $\pm 3.40 \times 10^{38}$
`double`	32/64	as `float` or `long double`
`long double`	64 bits	$\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$

Floating point numbers are represented in binary floating point form. The size of `double` is 32 bits by default. This can be changed to 64 bits by using the command-line option `--64bit-doubles`. The runtime library comes both in variants compiled with `double` set to 32 bits as well as 64 bits.

Subnormal numbers, infinity and NaN (not a number) are supported. The ranges stated in the table are for normal floating point numbers. Subnormal floating-point numbers extends the range of the exponent further at the cost of gradual loss of precision in the mantissa.

Floating point exceptions and changing the rounding mode are not supported.

32 bits format

In the 32 bits format the exponent is 8 bits and the mantissa is 23 bits. The precision is between 6 and 7 decimal digits.

64 bits format

In the 64 bits format the exponent is 11 bits and the mantissa is 52 bits. The precision is between 15 and 16 decimal digits.

11.1.6 Function pointer types

For the 68000 architecture function pointers are always 32 bits.

11.1.7 Data pointer types

For the 68000 architecture data pointers are always 32 bits.

Each data pointer has an associated index type, always a signed integer type. This type is used in address calculations, such as array access with an index or advancing a pointer by adding or subtracting an integer value.

11.1.8 Pointer conversions

Function and data pointers are treated as being unsigned values. In general, casting to a type that has fewer bits means a pointer value gets truncated. Casting to a wider type results in zero extension.

11.1.9 size_t

This unsigned integer type holds the maximum size of an object. On the 68000 the size of `size_t` is 32 bits.

11.1.10 ptrdiff_t

This signed integer type represents a distance within the largest possible object. On the 68000 the size of `ptrdiff_t` is 32 bits.

Subtracting two data pointers (within the same object) yields a `ptrdiff_t` value, representing the number of elements between the pointers, not the number of bytes.

Note: Standard C allows referring to an element one beyond the actual object. Subtracting the end address from the start address may result in a negative result if the value exceeds the ranfe of `ptrdiff_t`.

11.2 Structure types

Structure types are fully supported and can be nested. Structure members are stored sequentially in the order they appear in the declaration.

11.3 Union types

Union types are fully supported. The compiler also supports a useful extension that allows anonymous unions within a structure. Consider the following code:

```
struct Scope {
  union {
    char alpha;
    int num;
  } u;
  int b;
};

int main() {
  struct Scope x;
  x.u.num = 65;
  x.u.alpha = 'A';
  return 0;
}
```

The use of a declarator on the union within the structure requires an extra step to access its members. This was relaxed in the C11 standard, allowing you to write:

```
struct Scope {
  // Anonymous union
  union {
    char alpha;
    int num;
  };
  int b;
};

int main() {
  struct Scope x;
  x.num = 65;
  x.alpha = 'A';
  return 0;
}
```

This C11 extension, enabled by default, allows anonymous unions to be compiled without diagnostic messages, even though the compiler actually supports C99.

You can enable warnings for such extensions using the `-Wc11-extensions` or `-Wpedantic` command-line options.

You can also disable such extensions with the command-line option `--pedantic-errors`. In that case you will get an error instead.

11.4 Enumeration types

Enumeration types are represented as `int`. To use a smaller storage representation, choose a smaller integral type. For a better name, use a typedef:

```
enum fruit { apple, orange, banana };

typedef char fruit_t;

fruit_t active;

void citrus(void)
{
  active = orange;
}
```

11.5 Type qualifiers

Standard C provides two type qualifiers: `volatile` and `const`.

11.5.1 Volatile objects

C has the concept of volatile objects, typically used for hardware access. Both writing and reading volatile objects are considered side effects that will occur.

Related is the concept of sequence points in a program. A volatile access between two sequence points occurs between those points and cannot be moved past a sequence point. To ensure ordered memory accesses, make them volatile and separate them with a sequence point. The semicolon after a statement is an example of a sequence point:

```
uint8_t volatile * mem1;
uint8_t volatile * mem2;

void foo () {
   *mem1 = 2;
   *mem2;
}
```

In this case the write to `mem1` is guaranteed to be performed before the read of `mem2`. The read of `mem2` will also occur even if the result of the read is not used.

If multiple volatile accesses are done between two sequence points the order they happen in is undefined:

```
uint8_t volatile * mem1;
uint8_t volatile * mem2;

int foo () {

   return *mem1 + *mem2;
}
```

In this case both `mem1` and `mem2` are read, but the order in which they are read is undefined.

Access size

Accessing a volatile object wider than the natural register size on the target results in access performed in several steps, as dictated by the natural register size.

Reading and using only a portion of a scalar volatile object still results in the entire object being read:

```
volatile uint64_t wide;

uint16_t foo () {
   return wide;
}
```

Here the volatile object is 64 bits, but we are only interested in the lower 16 bits. In this case all 64 bits are read, the upper 48 bits are then discarded and the function returns the lower 16 bits.

If `wide` was not volatile, the compiler may instead choose to only read the lower 16 bits of the 64-bit `wide` variable.

Assignment results

Assignments in C has an expression value. Consider:

```
volatile int var;

int foo () {
   return var = 4;
}
```

The assignment writes `4` to the volatile variable, but the function return value (either `4` or the value read from `var` after the assignment) is implementation-defined.

Avoid such constructs in your programs. Be more explicit about your intent. To force a read after the assignment:

```
volatile int var;

int foo () {
   var = 4;
   return var;
}
```

If you want to be sure the function returns `4`:

```
volatile int var;

int foo () {
   var = 4;
   return 4;
}
```

This clarifies the intent and ensures consistent behavior across C compilers.

Bit fields

Accessing volatile bit fields has undefined behavior. A bit field describes a subset of bits within its storage unit. Adjacent bits may be accessed depending on layout. Using volatile bit fields is discouraged.

Note: Rather than using bit fields, define normal scalar values so that they cover hardware registers, following the defined or intended size of hardware register access. Then use expressions to extract or manipulate the part of the register you want.

11.5.2 Const objects

The `const` type qualifier indicates a read-only object. It can be applied to data objects and pointers.

When a static object is defined as `const`, the compiler attempts to place it in a read-only memory section.

A pointer to a `const` object can point to both read-only and writable objects. Such a pointer signifies that its user should not attempt to alter memory. This can aid the optimizer and is considered good practice, as it prevents unintended data alteration by parts of the application.

11.6 Type definitions

Using type definitions (the `typedef` keyword) is highly recommended, offering several benefits:

1. Code becomes more readable with meaningful type names. For instance, `speed_t` is clearer than `long`. Should the type definition change, only one location requires modification, preventing the need to search and selectively update `long` instances that represent speed.
2. A `struct fish` can be concisely named `fish_t`, which also hides its structure, if desired.

`typedef` incurs no cost; the generated code remains identical.

11.7 Alignment

The 68000 imposes even alignment for data objects larger than one byte. Padding between structure members are used if needed.

CHAPTER 12

Extended attributes

An attribute is a property attachable to functions, data objects, or types, specified as a keyword. Standard C includes built-in keywords like `const` and `volatile`.

Extended attributes provide access to behaviors or properties beyond Standard C. They are either target-specific or useful for embedded systems.

12.1 Overview

Attributes can be applied with either keyword syntax (e.g., `__far`) or attribute syntax (e.g., `__attribute__((far))`). Both are functionally equivalent, but the C parser may not accept the keyword form in some situations.

Usage is largely a matter of preference. Preprocessor macros can rename attributes for portability, enabling them to be toggled off for other targets or renamed to match different compilers or targets.

Note: The C parser sometimes produces unexpected errors with the keyword form of attributes (e.g., `__far`). If this occurs, use the `__attribute__((far))` form instead.

12.2 Using attributes

Type attributes can be applied to type declarations, following the same syntax as type qualifiers like `const` and `volatile`.

12.2.1 Syntax for data objects

You can apply attributes to data objects as follows:

```
__attribute__((far)) int a, b;
int __far c, d;
```

When applied to an object, the attribute's location is irrelevant. The example above applies the Far memory attribute to all defined objects (`a`, `b`, `c`, and `d`).

12.2.2 Syntax for pointer types

Attributes can also be applied to pointer types, where their location is significant. This determines whether the pointer itself is constant or what it points to is constant.

The easiest way to decipher attributes in function types is to read the type from right to left.

```
int __attribute__((far)) * p1;
long * __attribute__((far)) p2;
```

Here, `p1` is a pointer stored in default memory that points to an `int` in Far memory memory. `p2` is a pointer stored in Far memory memory that points to a `long` in default memory.

12.3 Attribute reference

This section goes through all available extension keywords and attributes.

12.3.1 Summary of attributes

The following table summarizes available attributes. For the keyword form, prefix with two underscores (e.g., `__far`). For attribute syntax, use the attribute name with `__attribute__` (e.g., `__attribute__((far))`).

Table 12.1: Extended attributes summary

Attribute name	Description
`aligned(nn)`	Specify alignment of data object, function or a record member
`section("name")`	Specify section name to use for a data object or function
`near`	Control storage of data object to near area
`far`	Control storage of data object to far area
`interrupt`	Used to define an interrupt function
`amiga_interrupt`	Amiga style interrupt function
`intrinsic`	Used to declare an intrinsic function
`saveds`	Entry point that needs to initialize the base pointer in register `A4`
`task`	Relaxes preserving registers

12.3.2 Description of attributes

This section describes each attribute in detail.

aligned

This attribute can be applied to functions, individual record members and data objects to force a certain minimal alignment.

The `aligned` attribute takes an argument which is the alignment to use:

```
__attribute__((aligned(16))) struct sprite ship;
```

Note: Certain data types may impose an alignment by themselves. The actual alignment is choosen so that all alignment constraints are satisfied.

---

section

This attribute can be applied to functions, global and static data objects to control the name of the section it is placed in.

The `section` attribute takes an argument which is the section name to use:

```
// Place in vram
__attribute__((section("vram")))
const char tiles[256] = { .. };

__attribute__((section("trueCode")))
long foo () {
  return 42;
}
```

See Description of pragma directives for how you can specify a section for multiple functions, global and static data objects.

near

This specifies a data object or a pointer to a data object that resides in a Near area. Accessing a global or static object in this area saves two bytes for each machine instruction used compared to the Far area.

Register `A4` is reserved to hold a base pointer to this area.

far

This specifies a data object or a pointer to a data object that can reside anywhere in memory.

The code needed to access Far memory tend to be slightly larger compared to the Near memory.

interrupt

An interrupt function is meant to serve as an interrupt handler. The interrupt attribute has the following effects:

1. An interrupt function cannot take any parameters
2. An interrupt function will preserve all registers used
3. Leaving the interrupt function uses a different instruction sequence compared to normal functions
4. The interrupt attribute may optionally be given a vector address as an argument.

The interrupt vector specified as an argument to the interrupt attribute:

```
int counter;

__attribute__((interrupt(0x0064)))
void irq () {
  counter++;
}
```

Note: The vector argument is optional. Omitting it means that there will be no vector section entry generated for that interrupt function. You can also suppress all vector sections from being generated by using the `--no-vector-sections` command-line option.

amiga_interrupt

The `amiga_interrupt` attribute defines an interrupt function intended to be used with the Amiga operating system. The Amiga interrupt has the following effects:

1. An Amiga interrupt function cannot take any parameters
2. Follows the register convention of an Amiga interrupt, registers `D0`, `D1`, `A0`, `A1`, `A5` and `A6` are considered scratch registers.
3. The return type should be `int` and you normally want to return `0` to allow interrupt processing further interrupt chain.

Here is a simple example of how an Amiga interrupt definition may look:

```
int counter;

__attribute__((amiga_interrupt))
int irq () {
  counter++;
  return 0;
}
```

Note: There is no interrupt vector associated with an Amiga interrupt definition. You need to use the Amiga operating system calls to install the interrupt handler.

intrinsic

The intrinsic attribute is used to declare intrinsic built-in functions. This can only be done on intrinsic functions that is already known to the compiler. Normally you use this by including the `calypsi/intrinsics68000.h` file which contains all such valid declarations.

saveds

A `saveds` function sets up the `A4` base address upon entry and restores the previous value of `A4` when returning. This is useful for API functions called from another context where `A4` may point to another base area or used for another purpose.

task

A `task` attribute can be used on functions such as `main` which is the start of the application. You will not normally call such functions from any C code. In that case you can apply the `task` attribute which relaxes preserving registers that would otherwise be saved on the stack. This can save a little stack space and will make the application a tiny bit smaller.

CHAPTER 13

Language extensions

The Calypsi C compiler supports several C language extensions that can be convenient. However, consider their portability implications.

13.1 Overloaded functions

The overloaded functions support provides overloading of functions in C similar to C++. Overloading in C is introduced using the `overloadable` attribute. For example, you might provide several overloaded versions of a `sine` function that invokes the appropriate standard function computing the sine of a value with `float`, `double`, or `long double` precision:

```
#include <math.h>

float __attribute__((overloadable)) sine(float x) { return sinf(x); }
double __attribute__((overloadable)) sine(double x) { return sin(x); }
long double __attribute__((overloadable)) sine(long double x) { return sinl(x); }
```

The compiler calls the most suitable function based on argument types. Overloaded functions are name mangled as C uses a single global namespace.

13.2 Statement expressions

This GCC extension allows a statement to appear where an expression is expected. Use it to allow loops, switches, and local variables within an expression.

A compound statement is a sequence of statements enclosed by braces. In the example below, parentheses around the braces create a statement expression:

```
({ int y = foo (); int z;
   if (y > 0) z = y;
```

(continues on next page)

(continued from previous page)

```
  else z = - y;
  z; })
```

The last statement in a statement expression is an expression followed by a semicolon. In this case, the variable z is used as the result of the entire statement expression.

13.2.1 Safe macros

Statement expressions can also implement safe macros that evaluate their parameters only once.

Recall the well known max macro:

```
#define max(a,b)  ((a) > (b) ? (a) : (b))
```

When used, this expands the largest value twice, which may be undesirable if it has side effects or represents an expensive computation.

With a statement expression this can be expressed as:

```
#define maxint(a,b) \
    ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

While this solves the double-evaluation issue, it introduces other subtle problems. First, you need to specify the type (see also __auto_type extension below), and it may cause variable shadowing.

In most cases, inline functions offer a better alternative to statement expressions. Inline functions avoid subtle variable shadowing problems, result in more readable code, and generate equally efficient code when inlined. The minor caveat is that inline functions may not be inlined, while statement expressions are always expanded in place.

13.3 Auto type

The auto type extension allows specifying a type that is automatically selected, using the __auto_type keyword. The declaration must declare only one variable, whose declarator must be an identifier. The declaration must be initialized, and the variable type is determined by the initializer type.

Using ___auto_type, the “max” macro in the previous section can be written to select the type automatically:

```
#define max(a,b) \
    ({__auto_type _a = (a); \
      __auto_type _b = (b); \
      _a > _b ? _a : _b; })
```

13.4 Typeof

Refer to the type of an expression using `__typeof`. The syntax resembles `sizeof`, but semantically, it acts like a typename defined with `typedef`.

Similar to `sizeof`, there are two ways to write the argument to `__typeof`: with an expression or with a type. For example, with an expression:

```
__typeof (x)
```

This creates a type identical to the identifier `x` in the current context.

You can also use a typename as an argument:

```
__typeof (int *)
```

In this case, the type is simply a pointer to `int`.

Typeof can also be used to implement the "max" macro:

```
#define max(a,b) \
    ({__typeof (a) _a = (a); \
      __typeof (b) _b = (b); \
      _a > _b ? _a : _b; })
```

13.5 Generics

C11 style `_Generic` is now supported. This allows expressions to expand to different outcomes based on a controlling expression. For example:

```
extern void stringFunc(char*);
extern void otherStringFunc(char*);
extern void string4Func(char[4]);

#define F(X) _Generic(&(X),        \
  default: otherStringFunc,        \
  char**: stringFunc,              \
  char(*)[4]: string4Func,         \
  char const**: stringFunc,        \
  char const(*)[4]: string4Func    \
  )(X)

void foo(char *p) {
  F(p);
  F("foo");
  F("longer string");
}
```

The macro `F` calls different functions based on its input. For example, a string literal `"foo"` (3 characters plus null terminator) is treated as a `char` array of 4 bytes, leading to a call to `string4Func()`. The `p` parameter, an ordinary `char*`, results in a call to `stringFunc()`. Finally, `"longer string"` is treated as a `char` array with a length other than four, matching the `default:` alternative and resulting in a call to `otherStringFunc()`.

CHAPTER 14

Pragma directives

Pragma directives are a standard C mechanism allowing vendor-specific extensions to the C language while maintaining portability.

You can use pragma directives to control compiler behavior, such as suppressing warnings or placing objects in specific memory areas using custom sections.

Pragma directives are always enabled and can be used with either the `#pragma` preprocessor directive or the `_Pragma` preprocessor operator.

14.1 Pragma directives reference

This section details all available pragma directives.

14.1.1 Summary of pragmas

The following table summarizes the recognized pragma directives:

Pragma directive	Description
`#pragma clang section`	Control custom section for code or data
`#pragma message`	Generate custom warning
`#pragma GCC warning`	Generate custom warning
`#pragma GCC error`	Generate custom error
`#pragma clang diagnostic`	Control diagnostic messages inside a source file
`#pragma require`	Force inclusion of other module
`#pragma rtattr`	Define a runtime attribute

Note: Since Clang is the C front end, some pragmas are inherited and function similarly. This simplifies compiler switching, despite pragmas being vendor-specific extensions.

14.1.2 Description of pragma directives

section

The `#pragma clang section` directive assigns section names to functions, global, and static objects.

The compiler typically places global and static objects into predefined sections, but you can override this to control placement to a specific memory area.

The section names can be specified as:

```
#pragma clang section bss="myBSS" data="myData" rodata="myRodata" text="myText"
```

The section names can be reverted back to default name by supplying an empty string to the section kind, for example:

```
#pragma clang section bss="" data="" text="" rodata=""
```

The new section name applies to all functions, global and static objects that follow from the pragma directive.

You are not required to define a name for every section category. If you omit some, the most recently specified one is used.

The `section` attribute takes precedence over the `clang section` pragma directive; a section name specified with `__attribute__((section("myname")))` has precedence.

Note: Section names are not interpreted. For example, naming a section `.bss.mySec` does not mean it will be a BSS section name.

---

message

You can generate a custom warning with `message` pragma. In addition there are a couple of GCC variants of this supported:

```
// The following will generate warning messages
#pragma message "my own diagnostic message"
#pragma GCC warning "my own diagnostic message"

// This will give and error
#pragma GCC error "not supported"
```

diagnostics

You can control diagnostic messages in source code using pragmas, which is useful for temporarily disabling specific warnings in a section of code.

The pragma controls any command-line-configurable warning. Warnings can be set to ignored, warning, error, or fatal.

You can also push and pop the current warning state. This is useful when writing a header file that others will compile, as their warning flags are unknown.

In the example below, `-Wextra-tokens` is ignored for a few lines; diagnostics then revert to their previous state.

```
#if foo
#endif foo // warning: extra tokens at end of #endif directive

#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wextra-tokens"

#if foo
#endif foo // no warning

#pragma clang diagnostic pop
```

The push and pop pragmas will save and restore the full diagnostic state, regardless of how it was set.

require

The `#pragma require` directive ensures that a module is included at the link stage, even if no functions or data objects within it are called or referenced.

This is primarily for libraries that request code slices or objects. For example, the C library's heap system uses a `require` pragma in its `malloc()` module to enable heap initialization during system startup:

```
#pragma require __call_heap_initialize
```

rtattr

The `#pragma rtattr` directive defines a runtime attribute within a C source file.

This is primarily for building libraries with alternative modules. While runtime attributes can be defined using the `--rtattr` command-line option, defining them in the source file may simplify build rules.

```
#pragma rtattr myAttribute "someValue"
```

CHAPTER 15

Intrinsic functions

This chapter covers predefined intrinsic and built-in functions.

Intrinsics functions appear as ordinary calls but are special compiler constructs, emitting specific instruction sequences. To enable them, include the `calypsi/intrinsics68000.h` header file.

Built-in functions are similar, but they are not declared in `calypsi/intrinsics68000.h`.

15.1 Intrinsic functions reference

The `calypsi/intrinsics68000.h` file appears as follows:

```
/*****************************************************************************
 *
 * Copyright Håkan Thörngren
 *
 * This file is part of the Calypsi C library.
 * Permission to use with the Calypsi tool chain is hereby granted.
 *
 *****************************************************************************/

#ifndef __INTRINSICS_68000_H
#define __INTRINSICS_68000_H

#ifdef __CALYPSI_TARGET_68000__

typedef unsigned short __interrupt_state_t;
typedef void (*__return_address_t)(void);

__attribute__((intrinsic)) void __disable_interrupts(void);
__attribute__((intrinsic)) void __enable_interrupts(void);
__attribute__((intrinsic)) __interrupt_state_t __get_interrupt_state(void);
__attribute__((intrinsic)) void __restore_interrupt_state(__interrupt_state_t);
```

(continues on next page)

(continued from previous page)

```
__attribute__((intrinsic)) __return_address_t __get_return_address(void);
__attribute__((intrinsic)) void __set_return_address(__return_address_t);

__attribute__((intrinsic)) void __stop_processor(__interrupt_state_t);
__attribute__((intrinsic)) void __reset_processor(void);
__attribute__((intrinsic)) void __break_instruction(unsigned char);

#endif // __CALYPSI_TARGET_68000__

#endif // __INTRINSICS_68000_H
```

15.1.1 Summary of intrinsic functions

Table 15.1: Intrinsic functions summary

Function name	Description
`__disable_interrupts`	Disable interrupts
`__enable_interrupts`	Enable interrupts
`__get_interrupt_state`	Get the current interrupt state
`__restore_interrupt_state`	Restore to a previous interrupt state
`__get_return_address`	Get the return address of the current function
`__set_return_address`	Alter the return address of current function
`__break_instruction`	Generate a `BKPT` instruction
`__reset_processor`	Generate a `RESET` instruction
`__stop_processor`	Stop the processor
`__builtin_clz`	Count leading zeroes in an `int`
`__builtin_clzl`	Count leading zeroes in a `long`
`__builtin_clzll`	Count leading zeroes in a `long long`
`__builtin_signbit`	Get the value of the sign bit

15.1.2 Description of intrinsic functions

This section details each intrinsic function.

`__disable_interrupts`

Emits a machine instruction that disables interrupts by setting the interrrupt level to 7.

Note: This can only be done in supervisor mode.

`__enable_interrupts`

Emits a machine instruction that enables interrupts by setting the interrrupt level to 0.

Note: This can only be done in supervisor mode.

`__get_interrupt_state`

Returns the current interrupt state as a value of type `__interrupt_state_t`. This can be used in the following way:

```
int global;

void safe_increment () {
  __interrupt_state_t state = __get_interrupt_state();
  __disable_interrupts;
  global++;
  __restore_interrupt_state(state);
}
```

In the example the current state of interrupts is obtained and saved in `state`. The code then ensures interrupts are disabled and then do the real work. Finally, the interrupt state is restored to what it was when the function was entered.

Note: This can only be done in supervisor mode.

`__restore_interrupt_state`

Restores to a previous interrupt state, see example above.

Note: This can only be done in supervisor mode.

`__get_return_address`

Reads the return address of the current function from its the stack frame.

`__set_return_address`

Replaces the return address of the current function with the address found in the argument of `__set_return_address()`.

`__break_instruction`

Emits a machine instruction that causes a breakpoint exception. This takes a value between `0` and `7` which encodes one of the 8 available software breakpoints.

`__reset_processor`

Emits the `RESET` instruction which has the effect of generating a reset to all external devices. Execution continues with the next instruction.

Note: This can only be done in supervisor mode.

`__stop_processor`

Emits an `STOP` instruction which takes an argument that is the value to set the processor status register to. This stops execution until an interrupt of high enough priority is received.

Note: This can only be done in supervisor mode.

`__builtin_clz`

Count the number of leading zeroes in a provided `int` value. Returns an undefined result if input is zero.

`__builtin_clzl`

Count the number of leading zeroes in a provided `long` value. Returns an undefined result if input is zero.

`__builtin_clzll`

Count the number of leading zeroes in a provided `long long` value. Returns an undefined result if input is zero.

`__builtin_signbit`

Returns the value (`0` or `1`) of the signbit of its input.

CHAPTER 16

Preprocessor

This chapter briefly describes the preprocessor and covers commonly used macros.

16.1 Overview

The preprocessor adheres to Standard C, providing:

1. Predefined macros for inspecting the compilation environment, allowing application tuning based on compiler properties.
2. User-defined macros, specified via the command line or within C source files.
3. The ability to dump the preprocessor environment for inspection.
4. The ability to dump the source file after preprocessing.

16.1.1 Predefined macros

The following describes the macros which are the ones you are most likely to use in conditional builds.

`__STDC__`

This macro is set to 1, indicating that this compiler adheres to Standard C.

__STDC_HOSTED__

This macro is defined as 0, indicating a cross-compiler. You can test this as follows:

```
#if defined(__STDC_HOSTED__) && __STDC_HOSTED__ == 0
```

__STDC_VERSION__

This macro is set to 199901L to indicate that this compiler adheres to the ISO/IEC 9899:1999 standard.

__CALYPSI__

This macro is set to 1, indicating that either the assembler or compiler in use is from Calypsi.

__CALYPSI_CC__

This is set to 1, indicating that this is a Calypsi C compiler.

__CALYPSI_ASSEMBLER__

This macro is set to 1 when a Calypsi assembler is used.

__CALYPSI_DEBUG__

This macro is defined and set to 1 when compiling with debug information enabled, i.e. `--debug` or `-g` command-line option used.

__CALYPSI_VERSION_MAJOR__

This macro is set to the first number of the version.

__CALYPSI_VERSION_MINOR__

This macro is set to the second number of the version.

__BIG_ENDIAN__

This macro is defined if the target has big-endian byte order.

`__LITTLE_ENDIAN__`

This macro is defined if the target has little-endian byte order.

`__BYTE_ORDER__`

This macro defines the byte order, taking the value of `__ORDER_LITTLE_ENDIAN__`, `__ORDER_BIG_ENDIAN__`, or `__ORDER_PDP_ENDIAN__`.

It is usually simpler to use `__BIG_ENDIAN__` or `__LITTLE_ENDIAN__` macros for conditional inclusion of endian-dependent source code.

`__ORDER_LITTLE_ENDIAN__`

This macro is defined to 1234 to describe the byte order in memory for a little-endian target.

`__ORDER_BIG_ENDIAN__`

This macro is defined as 4321, describing the byte order in memory for a big-endian target.

`__ORDER_PDP_ENDIAN__`

This macro is defined as 3412, describing the byte order in memory for a PDP-endian target.

`__FILE__`

This macro expands to the name of the current source file.

`__LINE__`

This macro expands to the current line number in the source file.

`__DATE__`

This macro expands to the current date.

`__TIME__`

This macro expands to the current time of the day.

`__COUNTER__`

This macro expands to a number that increments every time the macro is used.

`__CALYPSI_TARGET_68000__`

Set to 1 when the target is the 68000 family.

`__CALYPSI_CORE_68000__`

Set to 1 if the selected core is 68000. You can test if the core is 68000 using `#ifdef __CALYPSI_CORE_68000__`.

`__CALYPSI_CORE_68010__`

Set to 1 if the selected core is 68010. You can test if the core is 68010 using `#ifdef __CALYPSI_CORE_68010__`.

`__CALYPSI_CORE_68020__`

Set to 1 if the selected core is 68020. You can test if the core is 68020 using `#ifdef __CALYPSI_CORE_68020__`.

`__CALYPSI_CORE_68030__`

Set to 1 if the selected core is 68030. You can test if the core is 68030 using `#ifdef __CALYPSI_CORE_68030__`.

`__CALYPSI_CORE_68040__`

Set to 1 if the selected core is 68040. You can test if the core is 68040 using `#ifdef __CALYPSI_CORE_68040__`.

`__CALYPSI_CORE_68060__`

Set to 1 if the selected core is 68060. You can test if the core is 68060 using `#ifdef __CALYPSI_CORE_68060__`.

`__CALYPSI_CORE_68080__`

Set to 1 if the selected core is 68080. You can test if the core is 68080 using `#ifdef __CALYPSI_CORE_68080__`.

`__CALYPSI_TARGET_SYSTEM_EMBEDDED__`

Set to 1 when the compiler is configured for an embedded system, generating code suitable for flash or ROM memory. You can test if the compiler is configured for an embedded system using `#ifdef __CALYPSI_TARGET_SYSTEM_EMBEDDED__`.

__CALYPSI_TARGET_SYSTEM_AMIGA__

Set to 1 if the compiler is configured for the Amiga target. You can test if the compiler is configured for the Amiga target using `#ifdef __CALYPSI_TARGET_SYSTEM_AMIGA__`.

__CALYPSI_TARGET_SYSTEM_A2560K__

Set to 1 if the compiler is configured for the Foenix A2560K. You can test if the compiler is configured for the Foenix A2560K using `#ifdef __CALYPSI_TARGET_SYSTEM_A2560K__`.

__CALYPSI_TARGET_SYSTEM_A2560U__

Set to 1 if the compiler is configured for the Foenix A2560U. You can test if the compiler is configured for the Foenix A2560U using `#ifdef __CALYPSI_TARGET_SYSTEM_A2560U__`.

__CALYPSI_TARGET_SYSTEM_TOS__

Set to 1 if the compiler is configured for the TOS target. You can test if the compiler is configured for the Amiga target using `#ifdef __CALYPSI_TARGET_SYSTEM_TOS__`.

__CALYPSI_CODE_MODEL_SMALL__

This macro is set to 1 if the compiler is configured to use the Small code model.

__CALYPSI_CODE_MODEL_LARGE__

This macro is set to 1 if the compiler is configured to use the Large code model.

__CALYPSI_DATA_MODEL_SMALL__

This macro is set to 1 if the compiler is configured to use the Small data model.

__CALYPSI_DATA_MODEL_LARGE__

This macro is set to 1 if the compiler is configured to use the Large data model.

__CALYPSI_DATA_MODEL_FAR_ONLY__

This macro is set to 1 if the compiler is configured to use the Far-only data model.

CHAPTER 17

Section reference

Compiler-generated code and data objects are organized into sections within the object file. Understanding these sections is crucial for writing linker placement files and interpreting compiler and linker list files.

17.1 Section overview

Code and data objects occupy space in target memory, either as program-bits (actual data written to memory) or no-bits (sections represented only by their size in the ELF object file).

The Calypsi C compiler tool chain has adapted names that trace back to the origins to the early days of computing. These may not always be the best possible names, but they are widely known and familiar to many.

Sections are broadly categorized into three groups: text, data, and bss. Text sections are read-only program-bits. Data sections are read-write program-bits. BSS sections are read-write no-bits.

Note: If you ever wondered about what bss section stands for, it originally is block started by symbol, a pseudo operation in an assembler for IBM 704 from the mid fifties. Some people suggest it is easier to remember as better save space, as it is a way to save space in the output file by just storing the size of the section without any explicit data bits.

17.1.1 Section names

Each section has a name, and multiple section fragments can share the same name. Section names serve two purposes:

- To provide a descriptive identifier for the section.
- To control memory placement through rules that reference these names.

17.1.2 Section types

Each section has a type: text, data, or bss. For embedded systems, rodata (read-only data) and no-init sections are also available. Read-only data sections are similar to text sections but identify data rather than executable code. No-init sections resemble bss but lack the memory-clearing mechanism.

17.1.3 Data sections

From the compiler's perspective, data sections are read-write program-bits sections, meaning they have initializer values that reside in memory. In a hosted environment, such sections are loaded into memory before program execution. This approach is unsuitable for programs stored in flash memory where execution is expected to begin immediately upon power-on.

To address this, the linker clones data sections. The initializers for the original data section are moved and placed in the clone. The clone has a section name with an "`i`" prepended (e.g., `idata` for `data`) and is intended for flash (read-only) memory.

To initialize data sections on power-on, a copy routine is inserted before the `main()` function. This routine copies initializers from flash to RAM, initializing data memory and clearing bss sections.

The linker generates a `data-init_table` describing data to be copied and cleared.

Note: For some targets, the Calypsi C compiler tool chain can cross-compile executables for OS loading. The `--hosted` option prevents data section cloning, avoiding duplication. However, the table-driven initializer is still needed for BSS sections in such environments.

17.2 Sections used by the compiler

The following sections are sections used by the Calypsi C compiler tool chain.

Table 17.1: Sections

Section name	Type	Memory kind	Description
code	text	ROM	Executable code
nearcode	text	ROM	Executable near code
znear	bss	RAM	bss (zero initialized) near data
near	data	RAM	Initialized near data
zfar	bss	RAM	bss (zero initialized) far data
far	data	RAM	Initialized far data
cfar	rodata	ROM	Constant far data
switch	rodata	ROM	Switch tables
inear	rodata	ROM	Near data initializers
ifar	rodata	ROM	Far data initializers
data_init_table	rodata	ROM	Data initializer table
reset	text	ROM	Reset vector, when used
heap	noinit	RAM	Heap memory, for `malloc()`
stack	noinit	RAM	User stack
sstack	noinit	RAM	Supervisor stack

The sections `inear`, `ifar` and `data_init_table` in the table above are linker generated.

Note: It is assumed that there are no ROM or flash in either the Near area. Constants placed in either of those are placed in a normal writable section and the `const` attribute only affects the type of the object.

Note: The table assigns sections to ROM and RAM for ROM-based applications that start on power-up. For hosted systems loading from storage to RAM, this distinction is not applicable. Always consider ROM-marked sections as read-only.

17.2.1 The vector section

An interrupt function will have an associated vector. This is a specially named section for the purpose of holding a single vector. The name looks something like `$$interruptVector_0x00000000`. The intended address of the vector is encoded in the section name and the linker recognizes these and will place the vector at the address specified. This is handled without the help of any section placement rules.

17.2.2 Section reference

The following goes through the available sections in detail.

`code`

Holds program code, address range `0x00000000-0xffffffff`. This section is intended to be placed in a flash or ROM memory.

`nearcode`

Holds program code, address range `0x00000000-0xffffffff` but limited to about 32K. Code in this section treats function calls as being reachable with the `BSR` instruction that takes a signed 16 bit offset, which means all calls are assumed to be reachable.

`znear`

Holds zero initialized (bss) data reachable using base relative from register `A4`, up to 64K in total.

`near`

Holds initialized data reachable using base relative from register `A4`, up to 64K in total.

`zfar`

Holds zero initialized (bss) data in the main memory, address range `0x00000000-0xffffffff`.

`far`

Holds non-zero initialized data in the main memory, address range `0x00000000-0xffffffff`.

`cfar`

Holds initialized constant data in the main memory, address range `0x00000000-0xffffffff`. This section is intended to be placed in a flash or ROM memory.

`switch`

Holds switch tables in the main memory, address range `0x00000000-0xffffffff`. This section is intended to be placed in a flash or ROM memory.

`inear`

Holds initializers for the `near` section. This section is created by the linker by cloning the `near` section provided by the compiler. This section is placed in the main memory, address range `0x00000000-0xffffffff` and needs to be placed in a flash or ROM. This section is not used when linking for a hosted system.

`ifar`

Holds initializers for the `far` section. This section is created by the linker by cloning the `far` section provided by the compiler. This section is placed in the main memory, address range `0x00000000-0xffffffff` and needs to be placed in a flash or ROM. This section is not used when linking for a hosted system.

`data_init_table`

This section is created by the linker and filled in with information to the C startup code on how to copy and clear memory regions to properly initialize the data object before giving control to `main()`. This section is placed in the main memory, address range `0x00000000-0xffffffff` and needs to be placed in read-only memory, such as flash or ROM.

CHAPTER 18

Runtime library

This chapter describes the supplied C runtime library, adapted from the Apache NuttX library.

18.1 Design considerations

The original Apache NuttX library is a collection of routines from the C standard library, POSIX standard, and other components common in real-time operating systems.

It is a scalable and highly configurable runtime library that works in both 8-bits and 32-bits environments.

The adapted version becomes a Standard C library, with POSIX and RTOS-style components either removed or disabled.

Apart from that, the following are the key changes:

- Replaced header files closely related to the compiler (e.g., `stddef.h`, `stdarg.h`, `setjmp.h`, `stdint.h`).
- Internal names are changed to start with either double underscores or a single underscore followed by a capital letter. This avoids namespace pollution, as such identifiers are reserved for the compiler vendor.
- The stub interface uses names prefixed by `_Stub` instead of `up_` for clarity and to avoid namespace pollution.

Note: The Apache NuttX library's configuration utility, designed for environmental tailoring, is not used here; the build system is different. This allows the library to be used without extensive configuration, as the compiler is known and target-board-specific matters are excluded. Some library functions have different versions to reduce memory footprint by making certain capabilities optional (e.g., formatters; see Library on a diet).

18.2 Using the library

The C library consists of header files that are immediately available to use by the compiler and can be included:

```
#include <stdio.h>

int main () {
  printf("Hello World!\n");
  return 0;
}
```

When linking, the C library does not need to be explicitly added on the command line, as the linker automatically finds the correct C runtime library by examining the settings used during C object file compilation:

$ ln68k main.o linker-rules.scm

C library variants matching different compiler settings are provided in the installation directory; the appropriate one is automatically selected.

The linker detects mixing object files compiled with different settings or incompatible third-party libraries. This relies on matching runtime attributes in object files. A mismatch results in a descriptive linker error.

18.3 Provided library files

The following table lists the provided ready to use library files:

Table 18.1: Library variants

Library name	Code model	Data model	Size of double	Target system
clib-68000-sc-sd.a	Small	Small	32 bits	Generic
clib-68000-lc-sd.a	Large	Small	32 bits	Generic
clib-68000-sc-ld.a	Small	Large	32 bits	Generic
clib-68000-lc-ld.a	Large	Large	32 bits	Generic
clib-68000-sc-fod.a	Small	Far-only	32 bits	Generic
clib-68000-lc-fod.a	Large	Far-only	32 bits	Generic
clib-68000-sc-sd-double64.a	Small	Small	64 bits	Generic
clib-68000-lc-sd-double64.a	Large	Small	64 bits	Generic
clib-68000-sc-ld-double64.a	Small	Large	64 bits	Generic
clib-68000-lc-ld-double64.a	Large	Large	64 bits	Generic
clib-68000-sc-fod-double64.a	Small	Far-only	64 bits	Generic
clib-68000-lc-fod-double64.a	Large	Far-only	64 bits	Generic
clib-68000-sc-sd-amiga.a	Small	Small	32 bits	Amiga
clib-68000-lc-sd-amiga.a	Large	Small	32 bits	Amiga
clib-68000-sc-ld-amiga.a	Small	Large	32 bits	Amiga
clib-68000-lc-ld-amiga.a	Large	Large	32 bits	Amiga
clib-68000-sc-fod-amiga.a	Small	Far-only	32 bits	Amiga
clib-68000-lc-fod-amiga.a	Large	Far-only	32 bits	Amiga
clib-68000-sc-sd-double64-amiga.a	Small	Small	64 bits	Amiga
clib-68000-lc-sd-double64-amiga.a	Large	Small	64 bits	Amiga
clib-68000-sc-ld-double64-amiga.a	Small	Large	64 bits	Amiga
clib-68000-lc-ld-double64-amiga.a	Large	Large	64 bits	Amiga

continues on next page

Table 18.1 – continued from previous page

Library name	Code model	Data model	Size of double	Target system
clib-68000-sc-fod-double64-amiga.a	Small	Far-only	64 bits	Amiga
clib-68000-lc-fod-double64-amiga.a	Large	Far-only	64 bits	Amiga
clib-68000-sc-sd-a2560u.a	Small	Small	32 bits	A2560U
clib-68000-lc-sd-a2560u.a	Large	Small	32 bits	A2560U
clib-68000-sc-ld-a2560u.a	Small	Large	32 bits	A2560U
clib-68000-lc-ld-a2560u.a	Large	Large	32 bits	A2560U
clib-68000-sc-fod-a2560u.a	Small	Far-only	32 bits	A2560U
clib-68000-lc-fod-a2560u.a	Large	Far-only	32 bits	A2560U
clib-68000-sc-sd-double64-a2560u.a	Small	Small	64 bits	A2560U
clib-68000-lc-sd-double64-a2560u.a	Large	Small	64 bits	A2560U
clib-68000-sc-ld-double64-a2560u.a	Small	Large	64 bits	A2560U
clib-68000-lc-ld-double64-a2560u.a	Large	Large	64 bits	A2560U
clib-68000-sc-fod-double64-a2560u.a	Small	Far-only	64 bits	A2560U
clib-68000-lc-fod-double64-a2560u.a	Large	Far-only	64 bits	A2560U

Note: Libraries for 68020, 68060 and 68080 are also provided, by replacing 68000 with 68020, 68060 or 68080 in the above listed libraries.

Note: In addition to Amiga and A2560U shown above, there are also libraries for TOS and A2560K provided.

Note: There is normally no need to specify the C library to use when linking. The linker is able to automatically pick the correct C library by inspecting the C object files.

18.4 Stubs interface

How do you provide functions such as `fopen()`, `fprintf()`, or `assert()` when using a cross-compiler that may not know the final execution environment of an application, or even have a file system?

Functions like `fprintf()` pass through several library layers, handling formatting, buffered I/O, and stream interfaces. Ultimately, they output characters to a display or file. This is managed by a simple stubs API defined in the `calypsi/stubs.h` header file.

18.4.1 Semi-hosting

Semi-hosting is the concept where the debugger implements operations, such as I/O, on behalf of the target. This involves using a single breakpoint where the target temporarily stops to exchange data with the debugger, which performs the actual operation on the host. After the operation, any return value is passed back, and execution resumes on the target.

The Calypsi C compiler tool chain includes a semi-hosted debug implementation of the stubs API within the standard C library. To enable semi-hosting, add the `--semi-hosted` command-line option to the linker.

File operations are performed in the host filesystem by the debugger in a directory specified by the `--semi-hosted-root` command-line option, which defaults to the current directory.

The standard streams `stdin`, `stdout` and `stderr` are fully supported. Depending on your IDE you may have a console window for these streams.

Semi-hosting can be used during development for tracing or prototyping before a real I/O system is implemented in the application. In a final application, you will most likely use target-specific stub functions, not semi-hosted variants.

Note: Even if you implement some stub actions, you can still link with `--semi-hosted` for partial semi-hosted support. This allows you to implement a file system while retaining `assert()` through the semi-hosted mechanism. The C library's semi-hosted stubs are compiled with `--weak-symbols`, so your implementations take precedence. Unimplemented stubs will use their weak semi-hosted variants.

18.4.2 Stubs header

The `calypsi/stubs.h` file looks as follows:

```
/*****************************************************************************
 *
 * Copyright Håkan Thörngren
 *
 * This file is part of the Calypsi C library.
 * Permission to use with the Calypsi tool chain is hereby granted.
 *
 *****************************************************************************/

#ifndef __INCLUDE_CALYPSI_STUBS_H
#define __INCLUDE_CALYPSI_STUBS_H

#include <calypsi/config.h>
#include <stddef.h>

/* Currently not implemented. */
#define __noreturn_function

/*****************************************************************************
 * Public Data
 *****************************************************************************/

#if defined(__cplusplus)
extern "C"
{
#endif

/*****************************************************************************
 * Debug interfaces exported by the architecture-specific logic
 *****************************************************************************/

/*****************************************************************************
 * Name: _Stub_open
 *
 * Description:
 *   Open a file.
 *   The oflag argument are POSIX style mode flags, e.g O_RDONLY which
 *   are defined in fcntl.h.
 *   This function is variadic as it optionally can take a mode_t that
```

(continues on next page)

(continued from previous page)

```
 *   are permissions, e.g 0666. If the file system does not handle
 *   permissions you can ignore that this function is variadic.
 *   The return file descriptor shall be a positive number, larger
 *   than 2 (as 0-2 are used for stdin, stdout and stderr).
 *   The actual number does not matter and they need not to be
 *   consequtive, multiple numeric series with gaps between can be used.
 *
 * Return the obtained file descriptor or the desired errno value negated
 * if there is an error.
 *
 ******************************************************************************/

int _Stub_open(const char *path, int oflag, ...);

/******************************************************************************
 * Name: _Stub_close
 *
 * Description:
 *   Close a file
 *
 * Return 0 if operation was OK or the desired errno value negated
 * if there is an error.
 * Note: This will only be invoked for streams opened by _Stub_open(),
 *       there is no need to check for the standard descriptor 0-2.
 *
 ******************************************************************************/

int _Stub_close(int fd);

/******************************************************************************
 * Name: _Stub_lseek
 *
 * Description:
 *   Change position in a file
 *
 * Returns the new position in the file in bytes from the beginning of the
 * file, or the desired errno value negated if there is an error.
 *
 ******************************************************************************/

long _Stub_lseek(int fd, long offset, int whence);

/******************************************************************************
 * Name: _Stub_fgetpos
 *
 * Description:
 *   Get current position in a file
 *
 * Returns 0 on success, or the desired errno value negated if there is an error.
 *
 ******************************************************************************/

int _Stub_fgetpos(int fd, fpos_t *pos);

/******************************************************************************
 * Name: _Stub_fsetpos
```

(continues on next page)

(continued from previous page)

```
 *
 * Description:
 *   Change position in a file
 *
 * Returns 0 on success, or the desired errno value negated if there is an error.
 *
 ******************************************************************************/

int _Stub_fsetpos(int fd, const fpos_t *pos);

/******************************************************************************
 * Name: _Stub_read
 *
 * Description:
 *   Read from a file
 *
 * Returns the number of characters read or the desired errno value negated
 * if there is an error.
 *
 ******************************************************************************/

size_t _Stub_read(int fd, void *buf, size_t count);

/******************************************************************************
 * Name: _Stub_write
 *
 * Description:
 *   Write to a file
 *
 * Returns the number of characters actually written or the desired errno
 * value negated if there is an error.
 *
 ******************************************************************************/

size_t _Stub_write(int fd, const void *buf, size_t count);

/******************************************************************************
 * Name: _Stub_rename
 *
 * Description:
 *   Rename a file or directory
 *
 * Return 0 on success or the desired errno value negated if there is an error.
 *
 ******************************************************************************/

int _Stub_rename(const char *oldpath, const char *newpath);

/******************************************************************************
 * Name: _Stub_remove
 *
 * Description:
 *   Remove a file or directory
 *
 * Return 0 on success  or the desired errno value negated if there is an error.
 *
```

(continues on next page)

(continued from previous page)

```
 ****************************************************************************/

int _Stub_remove(const char *path);

/****************************************************************************
 * Name: _Stub_exit
 *
 * Description:
 *   Terminate the program with an exit code, exit clean-ups are done
 *   before this function is (finally) called.
 *
 ****************************************************************************/

void _Stub_exit(int exitCode) __noreturn_function;

/****************************************************************************
 * Name: _Stub_environ
 *
 * Description:
 *   Get the environment. On UNIX this is typically a global variable
 *   'environ', but in order to make it more flexible and avoid having
 *   such global variable (which is not part of the C standard) it is
 *   obtained using the stub interface.
 *
 * Note:
 *   This stub function is not implemented by the semi-hosted debug stub
 *   interface.
 *
 ****************************************************************************/

char** _Stub_environ(void);

/****************************************************************************
 * Name: _Stub_assert
 *
 * Description:
 *   Handle an assertion
 *
 ****************************************************************************/

void _Stub_assert(const char *filename, int linenum) __noreturn_function;

#if defined(__cplusplus)
}
#endif


#endif /* __INCLUDE_CALYPSI_STUBS_H */
```

18.4.3 Custom I/O

You can provide your own low-level I/O stubs. File descriptors are integers indexing an internal lookup table (typically an array). If the I/O system is used, the library statically allocates the first three streams: `stdin`, `stdout`, and `stderr`. Additional file streams are dynamically allocated from the heap.

18.4.4 Error handling

If a stub action encounters an error, it should return the negated `errno` value. The calling C library function then performs appropriate error handling, setting `errno` to the corresponding positive error code.

18.5 Examine use of library

When working with a memory-constrained system, the risk of running out of memory is always present. To help you understand memory usage, the linker can generate a list file with cross-reference information. This valuable tool details what is placed in memory, its location, and why it occupies that space.

You can tell the linker to provide a list file in the following way:

```
$ ln68k main.o clib-68000-lc-sd.a linker-rules.scm --list-file=project.lst --cross-reference
```

This will result in a list file names `project.lst` which contains several parts showing things such as:

- A summary of the memories
- An overview of sections and where they take space in the memories
- The object files used and from which library they have been extracted from
- Complete cross reference, showing where every section fragment is located with its size, together with:
  - What symbols are defined
  - What symbols are referenced
  - Who is referencing me
- An overall summary of total memory size

18.6 Library on a diet

If you use the C library in a memory-constrained system, you may find it rapidly consumes code space. This is because the C library provides substantial functionality, with many functions acting as simple facades to complex underlying implementations.

There are ways to tune things to reduce the memory footprint. However, first understand the current memory usage before making changes.

18.6.1 Formatters

The C library provides various `printf()` and `scanf()` variants, each with different capabilities and memory requirements.

While you can specify a format function, the compiler inspects used format strings and outputs this information to the object file. The linker then automatically selects the smallest variant compatible with those format strings.

Note: Automatic format function selection requires using string literals in calls. This is good practice, as it also allows the compiler to validate argument list consistency with the format string.

If you for some reason have to override the formatter used and dictate a specific one, you can do so using the `--rtattr` attribute which for `printf()` would be:

```
$ ln68k --rtattr printf=nofloat files...
```

The default print formatter as mentioned above, is selected automatically to match your needs. The following table describes the available print formatters:

Table 18.2: Print formatters

Capability	printf=reduced	printf=medium	printf=nofloat	printf=float
basics, `c d i o p s u X x %`	yes	yes	yes	yes
format flag, `0 + - #`	no	yes	yes	yes
field width	no	yes	yes	yes
long long	no	no	yes	yes
float, `e f g E F G`	no	no	no	yes

The `scanf()` function exits in three variants, mainly depending on if you want support for floating point numbers or not:

Table 18.3: Read formatters

Capability	scanf=medium	scanf=nofloat	scanf=float
basics, `c d i o p s u X x %`	yes	yes	yes
long long	no	yes	yes
float, `e f g E F G`	no	no	yes

Note: An appropriate default formatter for `scanf()` is automatically picked by the linker based on the needs of your application.

18.6.2 Reduced exit

Terminating an application involves closing open files and executing `atexit()` functions. For embedded applications, which may never exit, termination code can be redundant; closing files also adds related code. An implicit call to `exit()` occurs when `main()` returns. If proper exit code is undesirable, you can exclude it.

The simplest way to reduce exit code overhead is to add `--rtattr exit=simplified` to the linker command line. This prevents closing files or calling `atexit()` handlers. Instead, `exit()` will immediately jump to `_Stub_exit()`, the lowest-level termination routine.

18.6.3 Consider alternatives

If the library still occupies too much space, consider replacing certain functions with smaller, less flexible alternatives.

18.7 Override library functions

You may sometimes need to override a library function. Consider using an alternative function with a different name rather than replacing a standard one. Replacing a library function is appropriate if you have a more suitable application-specific implementation, especially when other library functions call it.

To replace a library function, add a new source file to your project using the same function name and prototype, then build your application normally.

Note: Function replacement works because library functions use weak symbols. Your replacement functions are non-weak by default, taking precedence over library functions during linking.

18.8 C startup

The initial system configuration is performed by the C startup object which is included in the standard C library.

The C startup is responsible for setting up the execution environment before giving control to the `main()` function. This typically includes setting up the stack pointer and providing initial values to static variables.

The C startup object also contains a couple of small optional sections that are only included if needed. They are responsible for initializing optional parts of the runtime system, such as the heap and the file streams.

If you do not use functions such as `malloc()`, the heap is not needed and the code to initialize the heap memory system is omitted. The same happens for file streams. If there are no file streams used, the code related to initializing and terminating them are omitted. This is done in order to reduce the memory footprint of the final application.

In case you want to study the source code of the C startup it can be found at `src/lib/lowlevel/cstartup.s` under the installation directory.

18.8.1 Customizing the startup

In many cases the default C startup will work fine without any changes. However, there are a couple of typical situations when some custom configuration is needed. One example is that the memory system needs to be configured immediately at power on, or if your application is going to run under an operating system that may impose special rules on initialization and termination.

If you only need to run some code to perform early initialization of the hardware you can provide your own `__low_level_init()` function which is called early in the C startup object, before any static C variables are given their start values.

If you need to make actual changes to the C startup object the easiest way is to copy the existing one to your project, make changes as required and include it in your build system.

To be able to properly suppress the C startup object in the C library you need to provide a different value for the `cstartup` runtime attribute in your own `cstartup.s`. The `cstartup.s` assembly source file starts with:

```
;;; C startup variant, change attribute value if you make your own
        .rtmodel cstartup,"normal"
```

You need to change the value `normal` to something else. What value you use does not matter so much, but using a descriptive value is probably a good idea:

```
;;; C startup variant, change attribute value if you make your own
        .rtmodel cstartup,"mycustom"
```

The modified C startup source file needs to be included in your build system. The linker also needs to be informed that it should use a specific C startup object, which is done using the `--cstartup` command line option:

```
% ln68k <object-files> <your-startup.o> clib-68000-lc-sd.a --cstartup=mycustom
```

Note: The C startup object has been carefully crafted to have a small footprint and to properly initialize the C runtime system. Even though it makes calls to handle the file system, initialize data areas and the heap, it is done in such way that the actual code is only included in the final application if these subsystems are used.

A good understanding of the tools and the provided C runtime is needed in order to make non-trivial changes to the C startup object. Incorrectly made changes may cause linker errors, unused subsystem being pulled in, or result in a runtime that is not properly initialized.

18.9 Time and date

The provided C library implements the functions and definitions in the `time.h` system header file. The time related types are defined as 64-bit types to avoid the year 2038 problem and reduce issues with overflow.

18.9.1 Providing a time

C defines two functions `clock()` and `time()` that provides a concept of a time. The library provides a default implementation for them that returns a value with all bits set, which means that the time is not available.

To use actual times you will need to provide your own implementation of `clock()` or `time()` and include it in your project.

The `CLOCKS_PER_SEC` macro is set to `1000` by the `stdint.h` header file.

18.10 Rebuilding the C library

The C library is designed to allow for a great deal of configuration without resorting to rebuild it. Situations where you may need to rebuild the C library are when you need a specific variant build, or to enable debug information to debug the library itself.

For the most part, building the library is straightforward, but there are certain files that needs to be built in certain ways, e.g. variants of `printf()`. To make it easier to build a variant C library there are build scripts provided in the `library-build` folder in the installation directory.

There is one build file for each library variant that are included in the installation. Select one that has a similar configuration as a starting point. It is a good idea to make copy of the build script and modify it to suit your needs.

CHAPTER 19

The optimizer

The compiler applies various optimizations to reduce code size and increase performance. This chapter describes how to control optimizations and highlights useful aspects, rather than detailing every optimization applied.

19.1 Overview

By default, the compiler performs basic rewrites, optimizes code selection for efficient instruction set utilization, and makes good use of the CPU's internal registers.

The resulting application is generally well-suited for debugging, as many optimization passes are disabled by default.

19.1.1 General settings

Optimizations can be broadly enabled using the following command-line options.

`-O1`

Enable some optimization passes.

`--O2`

Enable all provided optimization passes.

`--speed`

Allow application size to grow in order to make the application run faster.

`--space`

Tune optimizations more towards making the application small. This is the default.

19.2 Function inlining

Function inlining expands a function's body at its call site, typically increasing performance but potentially increasing code size.

Function inlining may be beneficial on code size for small functions as the inserted function body becomes tailored to the particular code surrounding it. Normally when making a function call there is a calling convention that has to be followed which puts restrictions on how registers are used.

Inlining is enabled by specifying at least `-O1`.

19.2.1 The inline keyword

The `inline` keyword in C is sometimes misunderstood. Its purpose is to expose a function body to many compilation units and hint that the function may be inline expanded instead of making an ordinary call to it. There is no guarantee that a function marked as `inline` is actually inlined, this decision is taken by the compiler. Futhermore, the compiler may also choose to inline functions that are not marked as `inline`.

19.2.2 Using the inline keyword

A function marked as `inline` is normally placed in a header file to allow it to be used by multiple compilation units:

```
inline max (int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

An inline function can also be marked as `static inline` in a similar way:

```
static inline max (int a, int b) {
  if (a > b) {
    return a;
  } else {
    return b;
```

(continues on next page)

(continued from previous page)

```
  }
}
```

You can use functions marked as `inline` or `static inline` in the same way as any other function. They can be called and in that case they may be subject to inline expansion and you can store it as a function pointer and make calls to it using that function pointer.

There are two cases when a function marked as `inline` or `static inline` requires a separate copy of the function. First, the compiler may choose to make an ordinary function call to it. Second, a function marked as `inline` may be stored in a function pointer.

19.2.3 Variants of inline functions

As hinted in the previous section, there are two kinds of inline functions, plain `inline` and `static inline`. As long as the compiler chose to do inline expansion these two variants work identically. They differ when the compiler choose to not inline expand it and make an ordinary function call instead.

`inline`

A function is marked as `inline` does not cause the compiler to generate a separate definition of the function. If such separate function is needed, it is assumed that one compilation unit provides the definition using the `extern` keyword:

```
extern inline max (int a, int b);
```

This should be placed in an ordinary C source file, not a header file.

In most cases you will need to tell the compiler to actually generate such separate function using an `extern` declaration somewhere. If you do not do this and the function is required by the application, a linker error will result that tells you that the function is not defined.

The main benefit of using `inline` rather than `static inline` is that you are ensured that there will be at most one separate definition of the function.

Note: One subtle benefit which `inline` has over `static inline` is that if you really must expand the function inline, then you can omit the `extern` declaration of the function. The linker will then let you know that the required inline expansion did not happen. In this case you most likely want to specify the command-line option `--always-inline` to ensure that the inliner is always used.

`static inline`

The compiler will generate a `static` standalone version of a `static inline` function whenever there is a need for it. Being `static` it is local to the compilation unit with no external linkage.

This means that if you make use of the same `static inline` function in several compilation units, there may be multiple copies of the same function.

Note: If you take the address of the same `static inline` function in different translation units then the result will not compare equal.

Note: While it is slightly easier to use a `static inline` function as you do not need to provide a single `extern` declaration, it comes with the cost of potentially having duplicate code in the final application.

19.2.4 Controlling inline expansion

The compiler will apply it own decisions on what to inline and when to use normal function calls. Small simple functions and static functions with only a single use are prime candidates for inline expansion.

There are a couple of command-line options that allow for tuning which functions are considered for inlining:

`--always-inline`

Always enable the inliner, no matter which optimization level is used.

This is useful if you have functions that you always want to be inlined.

`--no-inline`

Do not perform any inlining at all.

`--only-marked-as-inline`

Only consider functions that have the `inline` keyword. This prevents functions not marked as `inline` from being considered for inlining.

`--strong-inline`

Try to obey the `inline` keyword. This relaxes some of the requirements, such as that the function must be small enough. This is a strong hint that we really want functions marked for inlining to be inlined. The compiler may still refuse to inline a function.

Note: There are a number of reasons why the compiler may refuse to inline a function, e.g. used recursively, there is no prototype, it uses a variable argument list or it uses variable length arrays.

`--inline-on-matching-custom-text-section`

Functions being placed in a custom section (using `#pragma clang section`) are only allowed to be inlined into a function that belongs to the same custom section.

This is useful in situations when the memory system is being manipulated and functions are placed in a custom section to control placement to obey the memory setup. Especially if you have several such configurations you may want to prevent functions from being inlined cross custom sections as they may make assumptions on which section (memory placement) they are being executed from.

19.3 Cross call

This optimization pass can sometimes result in quite significant savings on the application size. It examines the almost final program looking for instruction sequences that are identical. Such sequences are lifted out to separate subroutines which are called instead. This takes in account the size of the code sequences as well as how many times they appear.

19.3.1 Controlling cross call

Cross call is enabled by specifying at least `-O2`. Even though cross call is focusing on the application size, it is enabled also when the `--speed` command-line option is specified, as the savings on code space can sometimes be quite significant.

`--no-cross-call`

This option disables the cross call optimization. This can be used with `-O2` when you want to avoid the subroutine call overhead that cross call will introduce.

`--no-interprocedural-cross-jump`

This option disables the cross jump between functions optimization. This can be used if you are manually mapping in functions by some bank mechanism in a way so that not all functions in the same compilation are visible at the same time.

CHAPTER 20

Assembly language interface

Assembly language provides symbolic access to target machine instructions. You might need this control for specific instructions unrepresentable in C, precise hardware interaction, exception stack frame manipulation, exact timing sequences, or performance-critical routines.

20.1 Intrinsic functions

The compiler provides intrinsic functions (declared in the calypsi/intrinsics68000.h file) that resemble ordinary functions. Instead of a function call, an intrinsic generates a specific instruction sequence. For example, `__disable_interrupts()` emits machine instructions to disable normal interrupts. See Intrinsic functions for details.

20.2 Assembly functions

You can implement functions in assembly language and call them from C like any other function. An assembly function must adhere to the C calling convention, which dictates how values are passed to the called function and where the return value is placed.

Note: To simplify your assembly routine interface, consider using the `simple_call` calling convention.

You can choose between a separate assembly source file or inline assembly; each has pros and cons. Assembly functions offer better separation between C and assembly, aiding portability. However, function call overhead and adherence to calling conventions may be undesirable.

Minimal boilerplate assembly code is required to place the routine in a suitable section and declare public symbols. This assembly code resides in a separate file, which must be added to the build system.

Assembly language files typically use the `.s` or `.asm` extension; however, this varies due to the lack of standardization in assembly language itself.

The assembler provided by the Calypsi C compiler tool chain, similar to UNIX assemblers, uses directives starting with a dot. This avoids name clashes with instructions, whose naming conventions vary widely across targets, ensuring consistent directive names.

As a minimum, you must declare the section and export your function name using the `.public` directive:

```
            .section code
            .public myFunction    ; export myFunction
myFunction: add.l   d1,d0         ; just add the inputs
            rts
```

To call this function correctly, you must provide a prototype in C:

```
extern int myFunction(int a, int b);

int caller(int x) {
  return myFunction(5, x);
}
```

20.2.1 Generate skeleton code

The easiest way to generate an assembly source file is to have the compiler create it using the `--assembly-source` command-line option. A simplified C source file containing the desired functions and declarations can be used for this purpose.

You can provide desired function definitions with simple parameter uses to study how they are passed:

```
extern int intvar;
extern char charvar;

extern void externalFunction(int*);

int myFunction(int i, char c) {
  int local = i;
  intvar = i;
  charvar = c;
  externalFunction(&local);
  return local;
}

int main() {
  myFunction(intvar, charvar);
  return 0;
}
```

```
$ cc68k skeleton.c --assembly-source=skeleton.s
```

```
; Generated by Calypsi ISO C compiler for Motorola 68000

            .rtmodel version,"1"
            .rtmodel nearDataBase,"A4"
            .rtmodel core,"68000"
            .rtmodel codeModel,"large"
            .rtmodel target,"none-specified"
            .extern charvar
            .extern externalFunction
```

(continues on next page)

(continued from previous page)

```
            .extern intvar
;  extern int intvar;
;  extern char charvar;
;
;  extern void externalFunction(int*);
;
;  int myFunction(int i, char c) {
            .section code,text
            .public myFunction
            .align  2
myFunction: subq.l  #4,sp
;    int local = i;
            move.l  d0,(sp)
;    intvar = i;
            move.l  d0,(.near intvar,a4)
;    charvar = c;
            move.b  d1,(.near charvar,a4)
;    externalFunction(&local);
            lea.l   (sp),a0
            jsr     externalFunction.l

;    return local;
            move.l  (sp),d0
;  }
            addq.l  #4,sp
            rts
;
;  int main() {
            .section code,text
            .public main
            .align  2
main:
;    myFunction(intvar, charvar);
            move.b  (.near charvar,a4),d1
            move.l  (.near intvar,a4),d0
            jsr     myFunction.l

;    return 0;
            moveq.l #0,d0
;  }
            rts
```

20.3 Calling convention

The default calling convention is complex in detail, but straightforward in most common scenarios. An alternative calling convention `simple_call` is also provided.

If parameters are passed on the stack, the caller is responsible for cleanup. The called function may use any register resource but must preserve certain registers, saving and restoring them before returning.

20.3.1 Simple calling convention

Use `__attribute__((simple_call))` or `__simple_call` on a function declaration to enable the simple calling convention.

In this calling convention all parameters are pushed on the stack. Registers `D0`, `D1`, `A0` and `A1` are destroyed by a call and the return value are passed as follows:

Table 20.1: Return values

Register	Size	Types
`D0.B`	8	`char`
`D0.W`	16	`short`
`D0.L`	32	`int`, `long`, `float`, data and function pointers
`D0:D1`	64	`long long`, `long double`

20.3.2 Normal calling convention

Parameters are passed in the `D0`, `D1`, `A0` and `A1` registers. These registers are clobbered by a function call. All other registers must be preserved by a function call.

Table 20.2: Parameter registers

Register	Size	Types
`D0.B`	8	`char`
`D1.B`	8	`char`
`D0.W`	16	`short`
`D1.W`	16	`short`
`D0.L`	32	`int`, `long`, `float`
`D1.L`	32	`int`, `long`, `float`
`A0`	32	data and function pointers
`A1`	32	data and function pointers
`D0:D1`	64	`long long`, `long double`

Parameters are bound to register left to right on a first fit basis. If a parameter register has to be skipped over, it will considered again for later parameters. Parameters that cannot be fit into registers are passed on the stack.

Table 20.3: Return values

Register	Size	Types
`D0.B`	8	`char`
`D0.W`	16	`short`
`D0.L`	32	`int`, `long`, `float`, data and function pointers
`D0:D1`	64	`long long`, `long double`

20.3.3 Structure passing

Structure parameters are passed on the stack. If a function returns a structure, the caller allocates space and adds an extra 'invisible' parameter (a pointer to that space) to the function call. The called function is expected to return this pointer.

20.4 Inline assembler

The inline assembler allows you to insert and interface assembly code slices within a C function. This avoids call overhead and can improve parameter adaptation. However, the optimizer must be more cautious, which may affect the performance of the surrounding C code.

20.4.1 Basic inline assembly

You can insert a slice of assembly code using an `__asm` block:

```
int counter;

void foo(char xx) {
  __asm(" bcc skip\n"
        " trapv\n"
        "skip: \n"
        );
}
```

Each line without a label requires at least one leading space and must be terminated by a newline character (\n).

The inline assembler supports the full assembly instruction set, including literal bytes, volatile operations, local labels, and register allocation for parameters and return values.

Goto labels and most assembler directives are currently not supported by the inline assembler. If you need better control with placement, use a separate assembly source file instead.

When inline assembly is inserted, the compiler adapts it to fit the generated C code. Variables can be passed as parameters, and a single result variable is supported. The compiler reasonably understands the inserted assembly, mixing it with C-generated code. Inline assembly is subject to low-level optimizations when the optimizer is enabled.

20.4.2 Volatile

An assembly block can be marked as `volatile`:

```
__asm volatile { ... }
```

This has the effect that all memory accesses in the assembly slice are treated as side effects. Otherwise the optimizer may remove reads from memory when the value read is not used.

20.4.3 Local labels

Local labels can be used and their names will not clash with C identifiers. When an inline assembly slice is inserted, local labels are converted to internal C labels, preventing name clashes with C identifiers.

20.4.4 External symbols

Inline assembly can refer to symbols defined outside its code slice, provided such symbols are visible at the C level within the same compilation unit.

20.4.5 Constraints

An inline assembly code slice can refer to C variables and return a value. The inline assembly construct optionally accepts three lists:

1. Output variable: Specifies a C variable to represent the returned value and a register class where the assembly code block places it. The result is prefixed by `=` in its single-value list.
2. Input expressions: Typically variables. The compiler evaluates the expression and places it in the specified register class.
3. Clobbered registers: Any register resource clobbered by the inline assembly must be specified here.

Multiple entries in a list are comma-separated.

Register classes

A register class is a register resource that can represent either a single register or a set of equivalent registers. They are used for allocating parameters and determining the return value location.

The following register classes are defined:

Table 20.4: Register classes

Register class	Description
`d0b`	`D0` as 8-bit register
`d0w`	`D0` as 16-bit register
`d0`	`D0` as 32-bit register
`a0w`	`A0` as 16-bit register
`a0`	`A0` as 32-bit register
`dreg8`	8-bit data register
`areg16`	16-bit address register
`dreg16`	16-bit data register
`xreg16`	16-bit address or data register
`areg32`	32-bit address register
`dreg32`	32-bit data register
`xreg32`	32-bit address or data register

Note: Register classes are used internally during code generation. Internal code generator rules ensure safe register allocation by adhering to specific rules and invariants. Rather than attempting to diagnose inline assembly constraints or impose conservative limitations, the compiler trusts you. Overusing resources or

violating internal invariants may lead to a register allocation error. In such cases, ease the register resources to find a working allocation.

Registers and constraints

The following code shows how constraints for an inline assembly code slice are defined:

```
int foo(short xx, int *p) {
  int out;
  __asm(" move.w (a0)+,d0\n"
        " ext.l d0\n"
        : "=Kd0" (out)
        : "Kd0w"  (xx), "Ka0" (p)
        : "d0", "a0"
      );
  return out;
}
```

Currently, the only supported constraint is 'K', followed by a register class.

An empty list can be entered by using a colon character followed by nothing.

The first list is the optional output parameter, describing the C variable where the output is visible after the inline assembly slice executes. The inline assembly slice must leave the result in the specified register class; the compiler will automatically insert code to store this value in the C variable.

The second constraint specifies input variables and their register classes. The compiler ensures these variables are available in the specified register classes before passing control to the inline assembly code slice.

The third list specifies the actual registers clobbered by the inline assembly code slice. These must be register classes describing a single register.

Substitutions

A register class may specify a register resource with multiple alternatives. The register allocator selects the actual register used. You can refer to the register resource using substitutions, given by the `%N` syntax, where N is the position of the result followed by the inputs, starting with `%0`.

A result always has position `%0`. If there is no result, the first input becomes `%0`. After this the inputs follow in order as `%1`, `%2` and so on:

```
int foo(int xx, int yy, char *p) {
  int out;
  __asm(" move.l %1,d0\n"
        " add.l %2,d0\n"
        " add.l (%3)+,%0\n"
        : "=Kd0" (out)
        : "Kdreg32" (xx), "Kdreg32" (yy), "Ka0" (p)
        : "d0", "a0"
      );
  return out;
}
```

If you find substitutions using numbers to be unreadable, you can specify a symbol for each substitution:

```
int foo(int xx, int yy, char *p) {
  int out;
  __asm(" move.l %[xx],d0\n"
        " add.l %[second],d0\n"
        " add.l (%[pointer])+,%[result]\n"
        : [result] "=Kd0" (out)
        : [xx] "Kdreg32" (xx), [second] "Kdreg32" (yy), [pointer] "Ka0" (p)
        : "d0", "a0"
      );
  return out;
}
```

CHAPTER 21

Assembler

The assembler enables writing assembly source files for C projects or standalone assembly language projects.

21.1 Overview

Unlike Standard C, there is no standard assembly language. The syntax for assembly instructions used by the 68000 assembler mostly follows what is outlined in guides made by the vendor of the instruction set.

Assembly source expressions are heavily inspired by C-style operators and precedence rules.

The assembler processes assembly source code using the C preprocessor, allowing C comments, include files, and conditional inclusion.

Directives are largely specific to the Calypsi C compiler tool chain, appearing consistent across products from Calypsi. Directive names begin with a leading dot, a common practice in many, but not all, assemblers.

Note: The dot prefix is used to prevent potential future conflicts where a mnemonic name might clash with a commonly used directive in the product line. This also makes directives stand out in the source code.

21.2 Syntax

21.2.1 Source file format

Assembly source lines follow the traditional format: a label field in the first column, followed by an instruction that may take one or more operands. Comments are preceded by a semicolon:

```
[label[:]]   [instruction  [operands]]  [; comment]
```

A label typically starts in the first column. If leading spaces are present, a label must be followed by a colon or enclosed in back quotes (see Symbol syntax below).

An instruction can be a mnemonic (target instruction name) or a directive. All directives begin with a leading dot as part of their name.

A label, a symbol describing a location, is defined by placing it in the first column of a source line, optionally followed by a colon.

Labels can also be declared by importing them with the `.extern` directive or by value using the `.equlab` directive.

Predefined words in instructions (mnemonics and operands) are case insensitive, while symbols are case sensitive. Some examples follow:

```
; Assembler comments start with a semicolon
;
Loop:          subq.l #1,d0     ; assembler comment
               nop
               BNE.S Loop
```

Predefined words are case-insensitive because both uppercase and lowercase assembly styles are commonly used.

21.2.2 Symbol syntax

Symbols are case sensitive and can be of arbitrary length. A symbol starts with a letter or underscore, followed by letters, underscores, and digits. Examples: `_4`, `a_symbol`, `abc`, `Test1`.

Any character is permitted in a symbol, provided the symbol is quoted with a back-tick. Examples: `` `another symbol` `` and `` `Table: 5` ``.

Intermixing symbols with the same name but different cases is possible, though considered poor style. Symbols in `as68k` are case-sensitive primarily to support the C compiler, which also has case-sensitive symbols. This also encourages consistent mixed-case symbols in assembly source code, which is considered good practice.

21.2.3 Preprocessor

The assembler uses a full-featured C preprocessor for input source files. It provides standard C preprocessor features, including header file inclusion, macro expansions, conditional compilation, and C-style comments.

For an introduction with examples on using the C preprocessor[4], refer to Wikipedia.

See Predefined macros for available macros, most of which are also relevant to the assembler.

21.3 Sections

The assembly source file is divided into sections using the `.section` directive. Each section, an indivisible unit of code or data, is laid out in memory by the linker according to rules from a placement rules file.

If no `.section` directive is specified, the assembler defaults to a `.section code` in the source file, meaning you start in a section named `code` unless otherwise directed.

Multiple sections enable more flexible placement of code and data by the linker than a single section. A section name can be used multiple times, with each instance creating an individual section fragment.

All sections are relocatable due to the ELF file format and must be defined in the linker rules file.

[4] http://en.wikipedia.org/wiki/C_preprocessor

21.3.1 Section kinds

The following table describes the supported section kinds. If unspecified, the section defaults to `text`.

Section kinds and modifiers are case-insensitive.

Section kind	Description
`text`	Executable code.
`data`	An initialized read/write data section in memory (RAM).
`rodata`	An initialized read-only data section in memory (ROM).
`bss`	"Block Started by Symbol"; holds zero initialized variables and variables not given an initializer value. The C runtime normally zero fill such area before calling `main()`.

21.3.2 Section modifiers

Section modifiers describe further behavior and can be specified as positive or negative.

Section modifier	Description
`noreorder`	Maintains the relative order of section fragments with the same name within a translation unit.
`reorder`	Allows section fragments of the same name to be placed arbitrarily, regardless of other fragments with the same name. This is the default.
`root`	Always include this section fragment in the program. This is the default for object files.
`noroot`	Only include this section fragment in the program if someone refers to a label inside it. This is the default for library files.

Note: Section fragments with the same name and the `noreorder` modifier are combined by the linker into a single placement group, ensuring contiguous placement. The absence of `noreorder` (or explicit `reorder`) allows the linker to place each section fragment arbitrarily, regardless of other fragments with the same name.

21.3.3 Section alignment

The `.align` directive specifies an alignment in address units. It advances the location counter and inserts fillers if needed, ensuring the next location has the specified alignment.

```
; Ensure that "table" label is placed at an address that can be
; evenly divided by 4.
        .section data
        .align   4
table:  ...
```

21.4 Expressions

Numeric expressions operate in signed (2-complement) mode. A range check occurs based on value usage; exceeding the possible range results in an error. The Operators table lists standard operators, supplemented by specialized relocation and section operators (see Relocation operators and Section operators).

You can optionally use spaces between values and operators in an expression.

Table 21.1: Operators

Operator	Precedence	Purpose
~	9	Unary bit-wise not
!	9	Unary logical not
-	9	Unary negate
+	9	Unary plus
*	8	Multiply
/	8	Divide
%	8	Modulo
+	7	Add
-	7	Subtract
<<	6	Bit shift left
>>	6	Bit shift right
>	5	Greater than
<	5	Less than
>=	5	Greater than or equal
<=	5	Less than or equal
==	4	Equal
!=	4	Not equal
&	3	Bit-wise and
^	2	Bit-wise exclusive or
|	1	Bit-wise or

21.5 Numeric constants

Integer constants can be entered in decimal, binary, octal, or hexadecimal. Decimal integers are sequences of digits not starting with zero. Prefixes `0x` indicate hexadecimal, `0b` binary, and `0` (followed by digits) octal.

Character constants are also supported, replaced by their ASCII value.

Some examples:

```
table:          .byte   0x1ff           ; hexadecimal number
                .byte   077             ; octal number (corresponds to decimal 63)
                .byte   65              ; decimal 65
                .byte   'A'             ; ASCII 65 (decimal)
                .byte   0b1011          ; binary (corresponds to decimal 11)
                .byte   0               ; zero (actually octal, but it is written
                                        ;       the same way in decimal)
```

21.6 Location counter

The assembler converts source programs into machine code for the 68000 processor. Instructions occupy consecutive memory locations until the next `.section` directive or end of file. The current instruction address is accessed by a single period (.), often called "dot".

The dot label, representing the current instruction location, is the location counter. It increments after each instruction, always holding the start address of the current instruction.

The location counter address is resolved by the linker but can be used in expressions. Short branches can use the location counter:

```
test:           tst.l   (10,a0)
                lda     (table),y
                bpl.s   .+4             ; skip next instruction of positive
                subq.l  #4,d0
```

However, using labels or local labels is often preferred (see below).

21.7 Local labels

Local labels are useful for local branch destinations in assembly source files and come in two variants, dollar-postfix alphanumeric and plus/minus sign character labels.

21.7.1 Dollar postfix style

Local labels end with a single `$`; the name can be an identifier or numeric (e.g., `loop$` or `3$`). A local label is active only between two non-local labels and cannot be exported to the linker. They serve as temporary locations for short-distance branching, typically for skips or local loops.

```
 loop$:         tst.l   (a0)+
                beq.s   15$
                subq.l  #1,d0
                bne.s   loop$
                bra.s   50$
15$:            move.q  #7,d0
```

After a non-local label, you can no longer refer to any local label before it. Consider if the following code is added below the one above, the presence of the non-local label `foo` resets the local labels:

```
foo:            subq.l  #1,d2
                bne.s   loop$           ; error, loop$ above no longer visible
```

As an alternative, you can use ordinary labels related to the context and add a number to make it unique. What you do is mostly a matter of taste.

21.7.2 Sign style

Local labels can also be created using sequences of `+` or `-` characters. The entire label name must use the same character, and its length is used for matching. References to labels with minus characters go backwards to the closest match, while plus characters go forward to the closest match.

This allows easy determination of whether the destination label is before or after an instruction. Sign-style local labels can pass over non-local labels.

```
+
                bne.s   +               ; this one goes to first + below
--
                beq.s   --              ; backward
+:              beq.s   ++++            ; goes over foobar
foobar:                                 ; I am not in the way
                nop
++++:
```

Note: Sign-style local labels are only usable with branch-style instructions. They are not allowed in more elaborate operands where an ordinary label is allowed.

21.8 Directives

All directives start with a single dot character.

The following table summarizes the directives known to the assembler:

Directive	Purpose
`.section` section-name, argument-list	Generate code for given section
`.equ` expr	Define a symbol value
`.equlab` expr	Define a symbol value that corresponds to a memory location
`.public` symbol-list	Export symbols to the linker
`.global` symbol-list	Synonym to `.public`
`.globl` symbol-list	Synonym to `.public`
`.pubweak` symbol-list	Export weak symbols to the linker
`.extern` symbol-list	Import symbol from other module
`.require` symbol-list	Require symbol from other module
`.incbin` filepath	Include a binary file as data
`.rtmodel` symbol, string	Define a runtime model attribute
`.byte` expr-list	Emits 8 bit values
`.word` expr-list	Emits 16 bit values
`.address` expr-list	Emits 24 bit values
`.long` expr-list	Emits 32 bit values
`.quad` expr-list	Emits 64 bit values
`.ascii` text	Emits the given string
`.asciz` text	Emits the given string with a terminating 0 (C string)
`.fillto` address [, value]	Fills memory with value
`.space` count [, value]	Fills memory with value
`.align` value	Specifies alignment
`.macro` parameter-list	Defines a macro
`.endm`	Ends a macro definition
`.end`	Stop processing the source file

A symbol-list is a list of symbols separated by commas. An expr-list is a list of expressions separated by commas.

21.8.1 Directives in detail

`.section`

The `.section` directive takes the name of the section as the first argument. It can optionally be followed by a kind and modifiers to describe the section further.

```
; A code section named "code" (text)
        .section code

; A code section named "code" (text) that are not stored
; in relative order to other "code" section fragments in
; the same compilation unit.
        .section code, reorder

; A data section named "storage"
        .section storage, data

; A constant area in ROM that are always included in output,
; even when put in a library (provided that anything
; in the compilation unit is referenced).
       .section table, rodata, root
```

.equ

Introduces a new symbol and gives it a value. The symbol appears in the label column and may have an optional colon after it:

```
BufNo       .equ  7
BufSize:    .equ  2 + ContentSize
```

Any expression can be used as a value, however using external symbols is subject to certain limitations imposed by relocations in the ELF object file format. In the case of valid expressions with external symbols, the value will be resolved by the linker.

.equlab

Constants can also be defined with the `.equlab` directive. This is similar to the `.equ` directive, with the difference that it defines a label, which describes a location, rather than a plain number:

```
MyLocation:   .equlab  0xD7
```

Where this matters is in the interpretation of the debugging information. Labels defined using `.equlab` are treated the same as other location labels. The debugger will understand that a symbol defined using `.equlab` can be used as a location when generating disassembly listings, while symbols defined using `.equ` will not be used for locations in the disassembly listing.

.public

Exports a symbol in the current file and makes in visible to other modules. Symbols are local to the file being assembled by default.

For shared definitions, an alternative is to put them in an include file and define them using the `.equ` directive.

.global

Synonym to `.public` for compatibility with other assemblers.

.globl

Synonym to `.public` for compatibility with other assemblers.

.pubweak

A weak symbol is created using the `.pubweak` directive in a similar way to `.public`. The difference is that a weak symbol may exist in multiple copies. Of these potentially multiple copies, one is selected by the linker. If there is a non-weak symbol among the weak ones, it will be picked by the linker.

Weak symbols serve a couple of purposes. They can be used for library replaceable objects where you can override a default library object using a non-weak public symbol. They are also useful for tools that generate assembly code where an identical construct may be generated multiple times, though only one is needed in the end.

.extern

Imports a symbol that is defined in some other module and make it visible in the current source file.

.require

A required symbol can be specified with the `.require` directive. It works similar to the `.extern` directive, with the difference that you do not need to actually use the symbol in any expression, it is being pulled in by the linker regardless.

This is mostly of interest when building modular software using libraries.

The typical use of this is that you have initialization code somewhere else built up using section fragments with the no-reorder property. The no-reorder property ensures that the code fragments appear next to each other. A section fragment is only active if someone actually refers to it. In this case the `.require` directive can be used to refer to it without actually using it. The result is that code which relies on that initialization code fragment exists can request that such code fragment becomes active.

.incbin

Include a binary file as embedded data directly into the program. This is useful if you have binary data assets as you can include them in the program without having to convert them to suitable source code.

```
my_data:
              .incbin  "rawdata.bin"
```

.rtmodel

It is possible to define runtime model attributes using the `.rtmodel` directive. Such attributes are checked at link time to ensure object file consistency. The attribute is an identifier and its value is a string:

```
; activities suitable for winter
              .rtmodel season, "winter"
```

Another source file could specify `season` to have another value:

```
; activities suitable for summer
              .rtmodel season, "summer"
```

If you try to link these two modules together will result in an error message describing that runtime model attribute `season` has a mismatch.

Source files that do not define the `season` attribute can be linked with either. It is also possible to use the special `*` value which also means it works with either. It is an explicit way of saying that I aware of this attribute and it works with whatever value it has:

```
; I can work with either season
              .rtmodel season, "*"
```

Functionally it is equivalent to not having the attribute defined. The difference is more in the eye of the reader, you have actively considered the attribute and concluded it works with whatever interpretation there may be.

Note: A defined runtime attribute affects the entire compilation unit it appears in. If you need to have a more narrow scope for some runtime model attribute, you need to break up the source file into smaller pieces.

`.byte`

Define one or more 8 bit values expressed as a list of comma separated expressions.

`.word`

Define one or more 16 bit values expressed as a list of comma separated expressions.

`.long`

Define one or more 32 bit values expressed as a list of comma separated expressions.

`.quad`

Define one or more 64 bit values expressed as a list of comma separated expressions.

`.ascii`

Take a string arguments and emit the bytes.

`.asciz`

Take a string arguments and emit the bytes followed by a terminating zero byte.

`.fillto`

Take an address offset and an optional filler value. The address value is relative to the start of the current section fragment. Emits the filler value until the location counter is equal address offset.

If the current section is bss, the filler value cannot be non-zero.

`.space`

Take an `count` value and an optional filler value. Emits `count` number of filler values in the output. If the current section is bss, the filler value cannot be non-zero and it only advances the location counter by `count`.

`.align`

Emit zero fillers to bring the location counter in alignment with the specified alignment argument. To long word align a table, use:

```
            .align 4
table       .byte  1,2,3,4
```

Note: The `.align` directive also have the effect of applying alignment on the section itself in the object file which forces the linker to place the section according to the alignment. If there are multiple `.align` directives with different alignment values in a section, the assembler will determine the overall alignment needed and emit that as the alignment of the section.

`.end`

End the source file and stop processing any further input. This directive is optional and processing will otherwise stop when the input file has been fully consumed.

`.macro`

Define a macro, see Macro language.

`.endm`

End a macro definition.

`.argdelim`

This directive defines delimiter characters that can be used include commas in an argument, see Macro language.

21.9 Relocations

The assembler and the C compiler produces relocatable output, which means that the program output does not have fixed addresses. The linker is responsible for placing sections at suitable memory addresses.

The compiler may need to refer to addresses that are unknown at compiler time. Such references are represented by relocations which are part of the object file. The linker processes the relocations after section placement to finalize the program.

Relocations are automatically generated by the assembler and C compiler when it encounters expressions or values that depend on final section placement.

21.9.1 Relocation operators

In certain contexts some additional prefix operators are available. They can be seen as relocation operators as they can be used to optionally introduce a relocation.

As they are relocations, they allow the operator to be executed at link time when the final addresses are known. However, they have the limitation that they must appear at the top level of the expression.

On the 68000 you do not normally need to load parts of relocatable addresses, but in some rare situations it may be useful. There are operator to extract 8, 16 and 32 bit portions of a larger value. They use operator names that is based on their width and the position, e.g. `.byte1`, `.word0`, `.word2` and `.long0`.

The number in the relocation operator refers to the least significant byte being extracted, which is followed by any additional bytes to make up for the size being extracted.

A value of `0` refers to the least significant byte part which is bit position `0`. A value of `1` refers to the byte that starts at bit `8`, and so on.

```
; Take a 16 offset in some imagined 64K page
            move.w   #.word0 table
```

Note: Relocation operators have high precedence like other unary prefix operators. If you want to access an address with an offset, you need to surround the expression by parentheses.

`.byte0`

The `.byte0` operator gives an 8-bit value from bit position 0 to 7 of an expression that is allowed to be resolved at link time.

`.byte1`

The `.byte1` operator gives an 8 bit value from bit position 8 to 15 of an expression that is allowed to be resolved at link time.

`.byte2`

The `.byte2` operator gives an 8-bit value from bit position 16 to 23 of an expression that is allowed to be resolved at link time.

`.byte3`

The `.byte3` operator gives an 8-bit value from bit position 24 to 31 of an expression that is allowed to be resolved at link time.

`.word0`

The `.word0` operator gives the lower 16 bits of an expression that is allowed to be resolved at link time.

`.word1`

The `.word1` operator gives the 16 bits from bit position 8 to 23 of an expression that is allowed to be resolved at link time. This mainly exists because it is useful for Foenix A2560 sprite data records.

`.word2`

The `.word2` operator gives the upper 16 bits of an expression that is allowed to be resolved at link time.

`.long0`

The `.long0` operator gives the lower 32 bits of an expression that is allowed to be resolved at link time. This is intended in cases where the value dealt with is a 64-bit value.

`.long4`

The `.long4` operator gives the upper 32 bits of an expression that is allowed to be resolved at link time. This is intended in cases where the value dealt with is a 64-bit value.

`.near`

This relocation gives the relative address of a symbol in the Near area, which corresponds to base pointer addressing using register `A4`. Typically it is given an expression to be resolved at link time. The expression is an address that is converted to a relative offset to the linker defined Near base symbol `_NearBaseAddress`:

```
0001                                    .extern foo
0002  00000000 202c....                 move.l  (.near (foo+2),a4),d0
```

21.9.2 Section operators

Section operators makes it possible to get hold of where a section is placed in memory.

Section names must be known to the assembler. If you want to refer to a section that is not used otherwise in the current assembly source file, simply declare it without inserting any code below it.

```
;;; Forward declaration
              .section elsewhere

              .section code
              ...
```

All these operators are resolved by the linker as placement is not known before link time. An error is given if a section spans multiple memories.

Table 21.2: Section operators

Operator	Precedence	Purpose
`.sectionStart` sectionName	9	The first address of the given section.
`.sectionEnd` sectionName	9	The last address of the given section.
`.sectionSize` sectionName	9	The size of the given section.

Note: Section size corresponds to `1 + end - start`.

Warning: If you allow the linker to intermix different sections in the same allocation range, these operators will base their values on the first and last section fragment of the given name. Different sections that are interleaved inside are silently included in the address range given by these operators.

21.10 Macro language

The `.macro` directive allows you to generate new commands that can create assembler output. A simple example follows:

```
foo           .macro  a, b
              .byte   \a
              .word   \b - 1
              .long   0
              .endm
```

This creates a new macro named `foo` which takes two arguments `a` and `b`. To use an argument inside the macro, prefix the parameter name with a backslash `\`.

21.10.1 Rules for argument substitutions

When looking for argument substitutions, the longest match is favored. This means if you have parameters called `a` and `aa`, substituting the longer name is always tried before shorter names. As there is no way to explicitly specify the end of a parameter name inside the body, a parameter may accidently try to match characters that comes after the parameter. A good rule of thumb is to make use of space to separate entities whenever possible. This also tends to improve readability.

21.10.2 Use of local labels

Each macro expansion will create a new unique context for local labels inside the macro body. Any previous local label context is restored after the macro is expanded. Thus, local labels inside a macro will not clash or interfere with any local labels surrounding the use of the macro.

This also works when using nested macro expansions. If a macro uses another macro inside its body, that inner macro expansion will have its own private local label context, and the previous context of the outer macro expansion will be restored when the inner macro has been expanded.

Thus, you are able to use the same label name inside a macro and in the code that uses it without any clash:

```
waitreg       .macro  reg
1$:           subq.l  #1,\reg
              bne     1$
              .endm

              move.l  #100,d0
              waitreg d0
1$:           move.l  #0,d1
```

Which would create the following list file:

```
################################################################################
#                                                                              #
# Calypsi assembler for Motorola 68000                          version 5.18 #
#                                                          15/Jul/2026  16:44:46 #
# Command line: example/macro-local.s -l                                       #
#                                                                              #
################################################################################

0001                    waitreg       .macro  reg
0002                    1$:           subq.l  #1,\reg
0003                                  bne     1$
0004                                  .endm
0005
0006  00000000 7064                   move.l  #100,d0
0007                                  waitreg d0
    \ 00000002 5380     `1$`:       subq.l  #1,d0
    \ 00000004 6600fffc             bne.w   `1$`
0008  00000008 7200     1$:           move.l  #0,d1

#########################
#                       #
# Memory sizes (decimal) #
#                       #
#########################

Executable  (Text): 10 bytes
```

21.10.3 Arguments with comma

Arguments to a macro are comma separated. This poses a problem in a situation where you want an argument to contain a comma character. The `.argdelim` directive defines a start and stop character that can be used to create an argument that contains a comma character.

```
              .argdelim <>
access        .macro  arg1, arg2
              ...
              .endm

              access  0, <2,a>
```

Here `arg1` is bound to `0` and `arg2` is bound to the value `2,a`.

The delimiter can be either one or two characters and you can pick any suitable character combination. By default there are no delimiter characters defined.

If a single character combination is not suitable, you can use two characters, e.g. `<-` and `->` which would be defined as follows:

```
              .argdelim <-->
access        .macro  arg1, arg2
              ...
              .endm

              access  0, <-2,a->
```

All delimiter characters are stripped and `arg2` is bound to `2,a` here as well.

CHAPTER 22

Running the assembler

The assembler is run from the command line, or from an IDE that interfaces with the assembler using the command line.

22.1 Basic invocation

The assembler is invoked in the following way:

```
$ as68k [options] sourcefile [options]
```

As an example, to assemble a source file with debugging information and a list file, you can use:

```
$ as68k --debug source.c -l
```

Command line options are optional arguments that tune the behavior of the assembler. They always start with a dash character. There are two variants, single letter options (`-l` to instruct the assembler to create a list file) starts with a single dash. The other variant is long descriptive options that starts with two dashes.

Some options require arguments. These follow the option, separated by a space or an equals sign (`=`). For single-argument options, the separator is optional:

```
$ as68k -Iinclude source.c -D VERBOSE=2 --list-file=tiny-source.lst
```

In this case the `-I` option adds `include` as a directory to scan for header files. The symbol `VERBOSE` is defined in the preprocessor with value 2. A list file with a specific name is also produced.

The order in which the options appear normally do matter, except for the `-I` option that adds directories to search for header files. Such directories are searched in the order in which they appear on the command line.

To display the version of the assembler, use `--version`:

```
$ as68k --version
Calypsi assembler for Motorola 68000 version 5.18
```

Command line options are described in Command line options.

22.2 Include search path

Header files are included by surrounding the filename with either double quotes or angled brackets:

```
#include "myheader.h"
#include <system.h>
```

Angled brackets are typically for system header files, while double quotes are for application-specific header files.

Note: The installation provides no system header files for the as68k assembler. However, the preprocessor retains its standard behavior of supporting angled include files and searching the installation directory.

The search order for include files is as follows:

1. Relative to compilation directory (double quote include only)
2. In directories specified in the `-I` option in the order they appear on the command line
3. The system directory, which is the Calypsi installation directory

Header files specified in angle brackets are not searched relative to the compilation directory. Otherwise, their behavior is identical.

Note: More precisely, the system header file directory is relative to the as68k executable. If you move an installation, it will still find the correct system header file directory.

Note: No header files are currently provided with the assembler. The preprocessor search path exists for consistency with the C compiler.

22.3 Assembler output

The assembler outputs one or two files, an object file that can be linked with other object files and libraries by `ln68k` to produce an executable file, and optionally a list file.

22.3.1 Object file

The object file is in ELF format, optionally including DWARF debugging information if `--debug` (or `-g`) is specified.

It contains:

- A symbol table
- Relocatable sections for code and data
- Relocations, allowing linker to modify address values after section placement
- DWARF-formatted source-level debugging information (if `--debug` was specified)

- Vendor-specific data, including runtime attributes and additional the Calypsi C compiler tool chain-specific information not covered by ELF/DWARF, used by the linker and debugger, with section types `0x8000000a` and `0x8000000b`.

Note: DWARF version 5 is used. The implementation covers the needs of the Calypsi C compiler tool chain and includes vendor extensions.

Note: DWARF debug information from the assembler includes source line information. The object file also contains a symbol table.

22.3.2 List file

The list file is a text file meant to be shown with a fixed width font. It contains a header that shows assembler, version, time when it was created and the command line used. The assembly source is shown with corresponding hex machine code. It also shows macro expansions and includes a summary of code and data sizes.

The following example is the `setjmp` source file from the C runtime library:

```
/*******************************************************************************
 *
 * Copyright Håkan Thörngren
 *
 * This file is part of the Calypsi C library.
 * Permission to use with the Calypsi tool chain is hereby granted.
 *
 *******************************************************************************/

              .rtmodel version, "1"
              .rtmodel cpu, "*"

              .public setjmp
#ifdef __CALYPSI_CODE_MODEL_SMALL__
              .section nearcode
#else
              .section code
#endif
              .align  2
setjmp:       move.l  sp,(a0)+
              move.l  a5,(a0)+
              move.l  (sp),(a0)+
              moveq.l #0,d0
              rts

#ifdef __CALYPSI_CODE_MODEL_SMALL__
              .section nearcode
#else
              .section code
#endif
              .public longjmp
              .align  2
longjmp:      move.l  (a0)+,sp
              move.l  (a0)+,a5
```

(continues on next page)

(continued from previous page)

```
            move.l  (a0)+,(sp)
            rts
```

Compiling with the `-l` option produces a list file:

```
################################################################################
#                                                                              #
# Calypsi assembler for Motorola 68000                          version 5.18 #
#                                                          15/Jul/2026  16:44:46 #
# Command line: setjmp.s -l                                                    #
#                                                                              #
################################################################################

0001                    /*******************************************************************************
0002                     *
0003                     * Copyright Håkan Thörngren
0004                     *
0005                     * This file is part of the Calypsi C library.
0006                     * Permission to use with the Calypsi tool chain is hereby granted.
0007                     *
0008                     *******************************************************************************/
0009
0010                                 .rtmodel version, "1"
0011                                 .rtmodel cpu, "*"
0012
0013                                 .public setjmp
0014                    #ifdef __CALYPSI_CODE_MODEL_SMALL__
0015                                 .section nearcode
0016                    #else
0017                                 .section code
0018                    #endif
0019                                 .align  2
0020  00000000 20cf     setjmp:     move.l  sp,(a0)+
0021  00000002 20cd                 move.l  a5,(a0)+
0022  00000004 20d7                 move.l  (sp),(a0)+
0023  00000006 7000                 moveq.l #0,d0
0024  00000008 4e75                 rts
0025
0026                    #ifdef __CALYPSI_CODE_MODEL_SMALL__
0027                                 .section nearcode
0028                    #else
0029                                 .section code
0030                    #endif
0031                                 .public longjmp
0032                                 .align  2
0033  00000000 2e58     longjmp:    move.l  (a0)+,sp
0034  00000002 2a58                 move.l  (a0)+,a5
0035  00000004 2e98                 move.l  (a0)+,(sp)
0036  00000006 4e75                 rts

#########################
#                       #
# Memory sizes (decimal) #
#                       #
#########################

Executable  (Text): 18 bytes
```

22.4 Command line options

This section covers the `as68k` command-line options in detail.

22.4.1 Options overview

Running the assembler from the command line without arguments results in a missing input file error, followed by a short help message:

```
$ as68k
Missing: FILE

Usage: as68k [--version] [-o|--output-file OUTPUT-FILE] [-l]
             [--list-file LIST-FILE] [-I DIRECTORY] [-D IDENTIFIER]
             [-U IDENTIFIER] [-g|--debug] [--rtattr NAME=VALUE] [--weak-symbols]
             [--core CORE] [--target TARGET] [--code-model NAME]
             [--data-model NAME] FILE
  use 'as68k --help' for detailed help
```

For more detailed help, use the `--help` option:

```
$ as68k --help
Calypsi assembler for Motorola 68000 version 5.18

Usage: as68k [--version] [-o|--output-file OUTPUT-FILE] [-l]
             [--list-file LIST-FILE] [-I DIRECTORY] [-D IDENTIFIER]
             [-U IDENTIFIER] [-g|--debug] [--rtattr NAME=VALUE] [--weak-symbols]
             [--core CORE] [--target TARGET] [--code-model NAME]
             [--data-model NAME] FILE
  use 'as68k --help' for detailed help

Available options:
  --version                Display version number
  -o,--output-file OUTPUT-FILE
                           Name of output file
  -l                       Generate a list file, named by appending '.lst' to
                           input file
  --list-file LIST-FILE    Generate list file, using given name
  -I DIRECTORY             Include directory
  -D IDENTIFIER            Predefine a macro
  -U IDENTIFIER            #undef a predefined macro
  -g,--debug               Produce debugging information
  --rtattr NAME=VALUE      Define a runtime attribute (identifier or quoted
                           string value accepted)
  --weak-symbols           Make all public symbols entries weak
  --core CORE              Core, one of '68000', '68010', '68020', '68030',
                           '68040', '68060' or '68080' (defaults to '68000')
  --target TARGET          Target system, one of 'Amiga', 'A2560U', 'A2560K' or
                           'TOS' (defaults to embedded/ROM use, if omitted)
  --code-model NAME        Code model, one of 'small' or 'large' (defaults to
                           'large')
  --data-model NAME        Data model, one of 'small', 'large' or 'far-only'
                           (defaults to 'small')
  -h,--help                Show this help text
```

22.4.2 Options in detail

`--version`

Displays the name and version of the assembler.

`--output-file, -o`

Specify the output object file name. If omitted, the output file is derived from the input file name (excluding path) with a `.o` extension, written to the current directory by default.

This option can also alter the output file name and provide a directory path. The specified directory must already exist.

`-l`

Generate a list file. The name used is the name of the input file (ignoring any directory path) with a `.lst` file extension. See also `--list-file`.

`--list-file`

Generate a list file. The name of the list file is given as argument to this option. See also `-l` to generate a list file based on the source filename.

`-I`

Add a directory to the current include search path. This option can be used multiple times on a command line. The order in which they appear specifies the search order between the directories.

The system include directory is always added last to the search order list.

`--include-system`

Add a directory to the current system include search path before the provided system include directories. This option can be used multiple times on a command line. The order in which they appear specifies the search order between the directories.

`--include-system-after`

Add a directory to the current system include search path after the provided system include directories. This option can be used multiple times on a command line. The order in which they appear specifies the search order between the directories.

`-D`

Define a macro. This takes an argument with the symbol name and optionally an assignment value `-Dsymbol[=value]`. If no value is given, the macro is given the value 1.

`-U`

Undefine a macro. This takes an argument with the symbol name to be undefined.

`--debug`

Generate DWARF symbolic debugging information. In order to get debugging information all the way to the debugger, the linker must also be given this option.

When debugging information is enabled the `__CALYPSI_DEBUG__` macro is also defined and set to 1.

`-g`

Synonym for `--debug`.

`--rtattr NAME=VALUE`

This defines a runtime attribute, written to the object file, that the linker can use for object file consistency checks.

`--weak-symbols`

Make all public symbols in the object file weak. This is normally not needed, but can provide a default library function implementation that is used if no replacement is provided.

In the assembler you can also specify symbols to be exported weak individually using the `.pubweak` directive.

`--core`

This selects which 68000 core to use. This can either be 68000, 68010, 68020, 68030, 68040, 68060 or 68080. If not specified it defaults to 68000.

`--target`

This option is available for symmetry with the compiler. It has the effect of setting the corresponding target preprocessor macro.

`--code-model`

This option is available for symmetry with the compiler. It has the effect of setting the corresponding code model preprocessor macro.

`--data-model`

This option is available for symmetry with the compiler. It has the effect of setting the corresponding data model preprocessor macro.

CHAPTER 23

Linker

The linker combines object files from the C compiler and assembler into an executable. Object files can also originate from libraries, such as the provided C runtime library.

Cross compilers need information about the target system, and these systems can differ significantly. A host linker, in contrast, is already tailored for its system.

23.1 Running the linker

Like other tools, the linker is command-line based and can also be used from an IDE. The IDE runs the linker via its command-line interface.

23.1.1 Basic invocation

The linker is invoked in the following way:

```
$ ln68k [options] [object-files] [library-files] rules-file
```

A rules file is required to describe the target. This `.scm` file contains information about the target system's memory and section placement rules.

Command-line options and input files can appear in any order.

Command-line options are optional arguments that tune linker behavior. They always start with a dash. Two variants exist: single-letter options (e.g., `-l` for a list file) begin with a single dash, while long descriptive options start with two dashes.

Some options require an argument, which appears after the option. It can be separated by a space or an equal (=) sign. For single-argument options, the argument may appear without a separator:

```
$ ln68k --debug file.o clib-68000-lc-sd.a placement.scm -o rocket.elf
```

In this example debugging information is retained in the output executable which is also named `rocket.elf`.

To display the version of the linker, use `--version`:

```
$ ln68k --version
Calypsi linker for Motorola 68000 version 5.18
```

23.2 Linking process

The goal of the linker is to combine object files produced by the compiler and the assembler. Object files have the file extensions `.o`. From now on we will refer to object files as "objects" as that is how they are seen by the linker once have been read from the file system.

The linker selects and combines objects in several stages:

1. Objects are selected for inclusion in the application.
2. Objects are placed at memory addresses by the linker following placement rules.
3. After placement the final address of each object is known. Using this knowledge the linker resolves relocations. This fixes all address references in the application.
4. The final program is emitted as an executable file.

23.3 Simplified placement rules

Understanding the linker process can be daunting due to its flexibility and numerous operations. Fortunately, it can often be used without in-depth knowledge, as it automatically derives placement rules from internal tool knowledge.

The linker cannot know available memory ranges or their types. This section describes placement in a simplified way; for more details, refer to Memory.

Note: On the Amiga you do not need to provide any linker placement rules as these are automatically generated internally.

23.3.1 Minimal rules

A minimal (or simplified) linker rules file primarily specifies memory locations:

```
(define memories
  '((memory flash (address (#x100000 . #x1fffff))
            (type ROM))
    (memory dataRAM (address (#x200000 . #x20ffff))
            (type RAM))
    (memory Vector (address (#x0000 . #x03ff))
            (section (reset #x0000)))
    ))
```

Each memory has a name, address range, and type.

The available types are shown in the following table.

Memory type	Description
RAM	This roughly corresponds to bss, and also includes the data area in an embedded system.
ROM	This is a combination of executable code, read-only variables, and everything else that should go into read-only memory, e.g. switch tables, string literals and constants.
ANY	This can be used in a hosted environment when the application is loaded into a single RAM memory. It combines allocation of RAM and ROM sections in the same memory range.
rodata	This type describes constants that are stored in read-only memory.
data	Initialized data area. Only exists when building for hosted use.
text	This type describes executable code.
bss	This is the data area for an embedded system or bare metal system. In a hosted system this holds the zero initialized data.

An error message will be given if the linker fails to automatically bind sections to memory, indicating the the sections which are causing problems. In such case you can bind those sections to a memory using a section rule. See Section placement.

In some cases you may have areas that need specific placement. This may be due to API jump tables or entry points. In such case you can explicitly place such sections using a section rule with an explicit address.

Note: If you have a special memory, such as a video memory that requires special access methods and cannot handle normal code or data, you may want to prevent the linker from doing automatic section binding there. This can be done by not specifying any type for it. In such case only sections with explicit binding will be placed in that memory.

23.4 ROM based systems

A ROM based system is typically an embedded or bare-metal system that runs without an operating system, taking control at power-on. The Calypsi linker assumes a ROM based system unless configured otherwise.

On a ROM based system, the application provides the reset vector. The C runtime configures itself by initializing static variables, setting up the stack, the heap (if used), and the C runtime itself. The I/O system is also initialized, but requires you to provide stubs for low-level stream-style I/O operations; see Stubs interface for more information.

If the hardware for your application has specific initialization needs, these must be provided by the application.

Initializing variables are done by clearing memory and filling in non-zero initializers using information stored in the ROM. The linker will split the initialized variables into two parts: the value part is stored in ROM and is copied to RAM during initialization before main() is called.

23.5 Host based systems

The alternative to a ROM based system is a hosted system that uses an operating system to load the application. Such operating systems vary from simple kernels to more elaborate systems.

Support for some hosted systems is provided, but given the vast number of alternatives, you may need to provide your own support.

The `--target` and `--hosted` command-line options configure the linker for hosted builds. Selecting a suitable application output file format is also part of this process.

When building for a hosted system, variables are initialized by loading their values in place. This avoids the duplication found in ROM based systems where initializers are stored in ROM and copied to RAM. The downside is that initialization occurs only once, during loading. If the application restarts, variables are not reinitialized. Linker options can override this behavior for detailed control, if needed.

See Target specifics for details about target specific support for supported hosts.

23.5.1 Simple hosted placement

The combination of an often simple executable file format and an application being loaded into RAM with both code and data gives certain problems. An application consists of both "bits" areas (code, tables and constant data) and "nobits" which is typically zero initialized data areas. In addition there are "data" areas where initializers are loaded in place, so they are a form of "bits" memory as well. Due to limitations in executable file formats, it is desirable to group "bits" and "nobits" separated, as the latter is often not represented in the executable file.

This can be solved by using a combined memory of type "any". The linker creates placement groups in that memory for different areas, one for each memory type. Placement is tailored for the executable file format.

Specific address placement

Using placement groups clashes with restricting a section to a specific address or limited memory area. If bits sections are placed first, then nobits sections, and there are also sections with address restrictions in the same memory, conflicts can arise.

To avoid conflicts, the linker forbids section rules with arbitrary restrictions when placement groups are used. However, some specific placement may be desirable, such as stub code placed first in memory. Such fixed address placement is allowed, but it must be specified in the memory that contains the placement groups.

23.6 Objects

This section looks at the objects being linked, how they are selected and how to control it.

23.6.1 C startup

In a C program the `main()` function starts the application. Before `main()` is called, code initializes the C runtime. This prebuilt system startup can be customized in rare cases; see C startup for details.

Although the C startup is a small module, it is intricate. Knowledge of how Calypsi handles its C runtime is necessary to modify it.

In general the C startup handles hardware initialization, stack setup, heap initialization and configuring the I/O system. It may also do things related to the `exit()` handling.

The startup code is written so that subsystems which are not used by the application are omitted. This selection process is a combination of the C startup, the linker and the C library.

23.6.2 Libraries

Multiple object files can be combined into a library file with a `.a` extension. This extension stands for archive, the common name for link libraries on UNIX.

The standard C library is provided as a library file.

Note: There are several variants of such library files provided to handle different compiler settings. You will normally not need to specify it as the linker in almost every situation is able to pick a suitable library automatically based on the provided object files.

23.6.3 Selected objects

Objects are selected from object files specified on the linker command line. All mentioned object files are included in the linker process.

Libraries also contribute objects as needed. The linker includes only objects from a library that are actually used by the application, which helps reduce the application size.

23.6.4 Object selection process

The linker selects object based on a tree shake algorithm. This is a worklist algorithm that starts with a root object. The root object references one or more external symbols. These symbols are placed in the work list. The linker will then process the first symbol in the work list and try to fulfill it using the available objects. Doing so may produce further external symbol which are placed in the work list unless they have already been processed.

This process continues until the work list is empty. If there are unresolved symbols they will be emitted as undefined symbol error messages.

When all symbols have been processed, the linker knows which objects are needed for the application.

23.6.5 The root

The actual root is a symbol named `__program_root_section` which identifies the section that must be included first. For a ROM based bare metal system, this is typically the section that holds the reset vector. For an operating system based system startup, it may be some stub or simply the section that contans the first executable code.

Note: If you study the system startup code you will find the `__program_start` symbol which can be seen as a close relative to the `__program_root_section` symbol. The purpose of the `__program_start` symbol is to tell the debugger about the first executable instruction of the application. It has no special meaning to the linker.

23.6.6 Placement

The actual placement of objects is described in Section placement.

23.6.7 Relocations

Consider a function in your application. This is described by a symbol with the same name as the function. In order to make a function call, the compiler needs to emit an instruction to transfer control to the function. How can the compiler know the address of the function when it does not know where the function is located? The answer is that it emits a relocation, which instructs the linker to adjust the application code when the final address is known.

Somewhat simplified, the relocation describes a location inside the object code that has to be altered. The called function is mentioned by its symbol. With this relocation information, the linker can figure out the location in the application to alter, and what to put there, to allow the actual call to be correctly made.

Relocating the program can be done by the linker once all objects have been given their final locations in the memory space of the application.

Note: In some supported operating systems relocations are only done partially by the linker and a further relocation is done when the operating system loads the application to memory.

23.7 Memory

When describing the target system, the linker uses memories. A memory is a named entity specifying a continuous range of storage locations. Memory descriptions have the following attributes:

name The memory name identifies a particular memory area.

address range The start and end addresses (inclusive) specify the memory's address range.

section Names of sections to bind to the current memory.

placement group A memory can only hold sections of one type; mixing code with zero- initialized data within a single memory is disallowed. In some output formats, it may be desirable to place different types of areas immediately adjacent within a single memory range. Placement groups function as sub-memories within a parent memory, each capable of holding sections of a distinct type. For example, one placement group can hold code sections, another data sections, and a third BSS sections. The

parent memory defines the overall range, and placement groups allocate memory within this range flexibly, allowing one group to begin immediately after the previous one concludes.

type Specifies which section types can be placed in this memory. See Minimal rules for more information.

qualifier Defines the address spaces that can be placed in this memory.

fill word Value used to fill unused locations in the memory, defaults to zero.

23.8 Linker rules file

The rules file (extension `.scm`) describes the available memory areas and defines rules for section placement. Sections can be explicitly bound to memory areas, or implicitly placed by omitting a binding rule. Implicit placement requires specifying a type for the memory; see Minimal rules.

You can often use a simplified rules file as described in Simplified placement rules. This section details linker rules. These rules can be elaborate and often unnecessary. You can intermix simplified and detailed rules to gain control without over-complicating the file. This approach helps you balance memory and placement requirements.

A full rules file can look as follows:

```
(define memories
  '((memory flash (address (#x100000 . #x1fffff))
            (section (nearcode (#x100000 . #x10a000))
                     code inear ifar cfar switch data_init_table))
    (memory NearRAM (address (#x200000 . #x20ffff))
            (section znear near))
    (memory RAM (address (#x210000 . #x23ffff))
            (section sstack stack zfar far heap))
    (memory Vector (address (#x0000 . #x03ff))
            (section (reset #x0000)))
    (block stack (size #x1000))
    (block sstack (size #x200))
    (block heap (size #x4000))
    (base-address _NearBaseAddress NearRAM #x8000)
    ))
```

The use of spacing for indentation here is unimportant to the linker tool, it is used it to make the file easier to read.

23.8.1 Section placement

Sections are bound to a specific memory by being mentioned after the `section` keyword. Placement of a section may be further restricted as described below.

Free placement

The `code` and `switch` sections appear alone (no surrounding parentheses). Sections with these names can be placed anywhere in the memory they belong to.

Restricted placement

The `nearcode` section appears with a specified range (`#x100000 . #x10a000`). If the range is larger than the size of the section, the section is place somewhere inside the allowed range.

Fixed placement

The `reset` section is placed at the fixed address `0x00000000` by having only a given address, not a range.

23.8.2 Block rule

Most sections are created by the compiler or the assembler and passed to the linker in the object file. It is also possible to create a section to represent a memory block in the linker rules file. Such blocks are used for the stack and the heap.

A block rule is defined among the memory rules. It has a name which is its section name and a size:

```
(define memories
  '((memory flash (address (#x8000 . #xffff))
    ...
    (block stack  (size #x800))
    (block heap   (size #x800))
```

You can omit mentioning blocks as the linker will attempt to create them automatically if left out. They are typically used for dynamic memory areas, such as heap and stack. There are some of these built in and the desired size can be specified on the command line. There is also a default size that the linker will fall back if it is not specified.

23.8.3 Stack size

The stack keeps track of the dynamic execution state, which includes how to return from function calls and local variables that are not held in registers. It also provides temporary storage during execution. The size needed depends on the application and the stack size can be defined in the linker rules files using a `block` rule:

```
(define memories
  '((memory RAM (address (#x210000 . #x23ffff))
            (section stack sstack zfar far heap))
    (block stack (size #x1000))
    (block sstack (size #x200))
   ))
```

Here the user stack size is set to `1000` hexadecimal (4096 bytes decimal) and is bound to the memory area named `RAM`.

The supervisor stack `sstack` is also needed on the 68000 and is here given the size `200` hexadecimal (512 bytes decimal) and is also bound to a memory area named `RAM`.

Note: You can override a stack size defined in the linker rules file using the `--stack-size` command-line option. This is often the preferred method.

23.8.4 Heap size

The heap is the area from which memory is obtained when using library functions such as `malloc()`. You only need to define a heap if your application actually uses functions such as `malloc()` or `calloc()`. It is perfectly fine for an application to run without a heap, and it may in some cases be desirable to do so.

If you decide to use a heap in your application you need to define its size in the linker rules file using a `block` rule:

```
(define memories
  '((memory RAM (address (#x210000 . #x23ffff))
            (section stack sstack zfar far heap))
    (block stack (size #x1000))          ; user stack
    (block sstack (size #x200))          ; supervisor stack
    (block heap (size #x4000))           ; heap
   ))
```

Here the heap size is set to `2000` hexadecimal (8192 bytes decimal) and is bound to a memory area named `RAM`. The text following a semicolon is a comment.

Note: You can override or define the heap size in the linker rules file using the `--heap-size` command-line option. This is often the preferred method.

23.8.5 Base address

A base address is a memory region permanently pointed to by a 68000 CPU register. Using base addressing for object access is typically more efficient due to optimized addressing modes on the 68000.

Defined in the linker rules file via a `base-address` rule, it works by associating a special symbol with the actual memory area it covers.

The 68000 defines one base addresses that describes a Near address area which allows for somewhat shorter instructions compared to full 32 bit addressing.

The `A4` address register is reserved to point to this area. As the 68000 has eight address registers and one is also the stack pointer, it leaves six address registers for general purpose use which is still quite generous.

The Near address area pointed to be the `A4` register is a single 64K bytes large memory area. This area is used in the Small data model for static and global data objects. In addition to this, you also have the stack and heap which are not allocated from this memory range.

The base address is defined in the linker rules file in the following way:

```
(define memories
  '(...
    (memory NearRAM (address (#x200000 . #x20ffff))
            (section znear))
    ...
    (base-address _NearBaseAddress NearRAM    #x8000)
    ))
```

The `_NearBaseAddress` symbol is tied to the `NearRAM` memory using an offset of `8000` hexadecimal, meaning the base address is in the middle of the memory range. This is because the offset from the `A4` base register is a signed 16 bit offset. Here it is given the address `208000` hexadecimal.

In both cases the C startup module takes care of setting up the `A4` CPU register before the `main()` function is called.

23.8.6 Placement groups

ELF output format limitations prevent mixing section types within a single memory. However, placement groups allow flexible organization.

When an application changes, varying section sizes within a memory range can be cumbersome, requiring frequent adjustments to memory ranges. Placement groups resolve this by dynamically allocating within the overall parent memory address range.

A placement group acts as a sub-memory within a parent, each holding sections of a single type. Multiple placement groups of different types can exist within one parent memory. Sections are filled one at a time. Once a group is placed, the linker continues allocation from the next free address, maximizing memory utilization without manual range specification for each group.

You can define placement groups in the following way:

```
(define memories
  '(...
    (memory near-bank (address (#x10000 . #x1ffff))
            (placement-group near-bits (section near cnear))
            (placement-group near-nobits (section znear)))
```

In this example the `near-bank` memory contains two placement groups `near-bits` and `near-nobits`. The `near-bits` placement group contains two sections `near` and `cnear`. The `near-nobits` placement group has a single section `znear` which is a zero initialized BSS section.

Note: The use of `near` and `cnear` makes sense when compiling for a hosted environment where a writable `near` section and a read-only `cnear` section can be placed next to each other as both are loaded into system RAM memory. On a ROM based embedded system they would go into separate memories.

Note: The names `near-bits` and `near-nobits` have no special meaning to the linker.

23.9 Scatter

In some situations you have a memory bound to a given address range at runtime, but that is stored at a different address in the application image.

Examples of this are an overlay system, a banking system, or a ROM that is located at some address, but parts of it is copied to RAM at a different location.

The linker implements this using scatter-to which is a property that can be given to a memory:

```
(memory bankSlotRAM
        (address (#xa000 . #xbfff))
        (scatter-to RAM-banks)
```

(continues on next page)

(continued from previous page)

```
        :generate-instances
        (section bankedcode))
```

In this case the bankSlotRAM memory has an address range A000-BFFF. Sections of type bankedcode are bound to it.

Using the scatter-to property, the entire memory is converted to a new section named RAM-banks. This section can be placed in another memory:

```
(memory bankedRAM (address (#x2000 . #x1fffff))
        (section RAM-banks))
```

The optional :generate-instances is useful when implementing an overlay system. It indicates that once the bankSlotRAM memory is full, a new empty instance can be generated to accommodate further bankedcode sections. Thus, memory is created on the fly and each instance is scattered to the bankedRAM memory.

If :generate-instances is not specified then only one instance of the memory is allowed.

Note: Relocations in the memory are done towards the runtime address, which is bankSlotRAM in the example above. This is the address area where the memory is intended to appear at runtime.

23.10 Linker list files

A list file can be generated from the linker using the --list-file (or -l) command-line option.

The list file is a valuable tool if you want to see how much memory your application actually needs. It can also answer questions about what objects are included from libraries, why they are included and where they are placed.

This is especially useful if want to understand how your application uses memory and can give insights to have it may be tuned.

As an example, consider the following example program:

```
#include <calypsi/stubs.h>

__task int main () {
  return 0;
}
```

This is intended to be a minimal program, but with debug support to allow the debugger to catch the call to exit().

If built and linked with the standard library and you specify the command-line options "-l --cross-reference --rtattr exit=simplified" you will get a list file which contains something like:

```
####################
#                  #
# Memories summary #
#                  #
####################

Name               Range             Size     Used     Checksum  Largest unallocated
```

(continues on next page)

(continued from previous page)

```
--------------------------------------------------------------------------------
Vector             00000000-000003ff 00000400   0.8%   none      000003f8
flash              00100000-001fffff 00100000   0.0%   none      000fffb2
NearRAM            00200000-0020ffff 00010000  12.5%   none      0000e000
  > NearRAM-NoBits 00200000-00201fff 00002000 100.0%   none      0000e000


####################
#                  #
# Sections summary #
#                  #
####################

Name   Range              Size     Memory         Fragments
-------------------------------------------------------------
reset  00000000-00000007  00000008 Vector         1
code   00100000-0010004d  0000004e flash          7
sstack 00200000-00200fff  00001000 NearRAM-NoBits 1
stack  00201000-00201fff  00001000 NearRAM-NoBits 1


###################
#                 #
# Placement rules #
#                 #
###################

Name             Address range     Key
-------------------------------------------------------------------------
NearRAM          00200000-0020ffff DATA
  NearRAM-NoBits 00200000-0020ffff BSS
    > stack
    > sstack
Vector           00000000-000003ff
  > (reset 00000000)
flash            00100000-001fffff TEXT and RODATA
  > code

Name   Size     Align
----------------------
stack  00001000 2
sstack 00001000 2

Name             Memory  Offset
---------------------------------
_NearBaseAddress NearRAM 00008000

################
#              #
# Object files #
#              #
################

Unit Filename          Archive
-----------------------------------------
  0  main.o            -
```

(continues on next page)

(continued from previous page)

```
          >  code 00000004
  2  cstartup.o        clib-68000-lc-sd.a
          # picked based on cstartup=normal (built-in default)
          >  code  0000002e
          >  reset 00000008
  5  simplified_exit.o clib-68000-lc-sd.a
          # picked based on exit=simplified (specified on the command line)
          >  code 00000008
  7  debug_exit.o      clib-68000-lc-sd.a
          # picked based on stubs=semi_hosted (due to configured to use semi-hosting)
          >  code 00000012
  8  debug_break.o     clib-68000-lc-sd.a
          # picked based on stubs=semi_hosted (due to configured to use semi-hosting)
          >  code 00000002

###################
#                 #
# Cross reference #
#                 #
###################

Section 'sstack'  placed at address 00200000-00200fff of size 00001000
(linker generated)

Section 'stack'  placed at address 00201000-00201fff of size 00001000
(linker generated)

__program_root_section in section 'reset'
 placed at address 00000000-00000007 of size 00000008
(cstartup.o (from clib-68000-lc-sd.a) unit 2 section index 8)
    Defines:
        __program_root_section = 00000000
    References:
        .sectionEnd(sstack)
        __program_start in (cstartup.o (from clib-68000-lc-sd.a) unit 2 section index 2)

__program_start in section 'code'
 placed at address 00100000-0010001b of size 0000001c
(cstartup.o (from clib-68000-lc-sd.a) unit 2 section index 2)
    Defines:
        __program_start = 00100000
    References:
        _NearBaseAddress
        .sectionEnd(stack)
        __low_level_init in (cstartup.o (from clib-68000-lc-sd.a) unit 2 section index 7)
    Referenced from:
        __program_root_section (cstartup.o (from clib-68000-lc-sd.a) unit 2 section index 8)

Section 'code'  placed at address 0010001c-00100029 of size 0000000e
(cstartup.o (from clib-68000-lc-sd.a) unit 2 section index 6)
    References:
        exit in (simplified_exit.o (from clib-68000-lc-sd.a) unit 5 section index 2)
        main in (main.o unit 0 section index 2)

_Stub_exit in section 'code'
 placed at address 0010002a-0010003b of size 00000012
```

(continues on next page)

(continued from previous page)

```
(debug_exit.o (from clib-68000-lc-sd.a) unit 7 section index 2)
    Defines:
        _Stub_exit = 0010002a
    References:
        _DebugBreak in (debug_break.o (from clib-68000-lc-sd.a) unit 8 section index 2)
    Referenced from:
        exit (simplified_exit.o (from clib-68000-lc-sd.a) unit 5 section index 2)

exit in section 'code'  placed at address 0010003c-00100043 of size 00000008
(simplified_exit.o (from clib-68000-lc-sd.a) unit 5 section index 2)
    Defines:
        exit = 0010003c
    References:
        _Stub_exit in (debug_exit.o (from clib-68000-lc-sd.a) unit 7 section index 2)
    Referenced from:
        (cstartup.o (from clib-68000-lc-sd.a) unit 2 section index 6)

__low_level_init in section 'code'
 placed at address 00100044-00100047 of size 00000004
(cstartup.o (from clib-68000-lc-sd.a) unit 2 section index 7)
    Defines:
        __low_level_init = 00100044
    Referenced from:
        __program_start (cstartup.o (from clib-68000-lc-sd.a) unit 2 section index 2)

main in section 'code'  placed at address 00100048-0010004b of size 00000004
(main.o unit 0 section index 2)
    Defines:
        main = 00100048
    Referenced from:
        (cstartup.o (from clib-68000-lc-sd.a) unit 2 section index 6)

_DebugBreak in section 'code'
 placed at address 0010004c-0010004d of size 00000002
(debug_break.o (from clib-68000-lc-sd.a) unit 8 section index 2)
    Defines:
        _DebugBreak = 0010004c
    Referenced from:
        _Stub_exit (debug_exit.o (from clib-68000-lc-sd.a) unit 7 section index 2)

#########################
#                       #
# Memory sizes (decimal) #
#                       #
#########################

Executable       (Text):   86 bytes
Non-initialized        : 8192 bytes
```

The linker list file displays how section fragments are distributed across memories and the total allocated memory size.

The `Memories summary` area summarizes the application's memories, including the size, usage, largest unallocated block, and an optional checksum of each memory.

The `Sections summary` area summarizes section placement and fragment counts.

The `Object files` area lists object files, their originating libraries (if any), and their section and overall size

contributions.

The `Cross reference` area provides detailed information for each code segment, including its entry point, name, address range, size, defined symbols, referenced symbols, and referencing code segments. This offers full bi-directional reference information for all application code segments.

Finally, the `Memory sizes` area summarizes the total memory used. Note that memory amounts are given in decimal here; elsewhere in the list file hexadecimal numbers are used.

23.11 Output formats

Additional executable output format can be selected with the `--output-format` command-line option.

Note: The linker will always produce an ELF output which is the format used by the `db68k` debugger.

23.11.1 ELF/DWARF

This is the base output format of the executable file. It contains an ELF program image of the executable application. If you specify `--debug` all DWARF debugging information sections found in the object files are passed on to the final executable image. The output file has file extension `.elf`.

23.11.2 Intel hex

The Intel hex file format contains the output binary as ASCII text. Output files have the `.hex` extension.

23.11.3 Motorola S-record

This is the Motorola S-record, sometimes called SREC. The output binary is expressed in ASCII text form. The output file has file extension `.srec`.

The S-record format has variants: S19, S28, and S37, corresponding to 16, 24, and 32-bit addresses. Using these formats forces the linker to output in a specific address size. The output file extension used corresponds to the format variant choosen, `.s19`, `.s28` or `.s37`.

23.11.4 Raw

The `raw` format represents program memory as a raw binary image file. Output files use the `.raw` file extension.

Note: In the default setting only one memory area can be emitted in Raw format. Multiple areas can be emitted by specifying the `--raw-multiple-memories` command-line option, this also has the effect of changing the name of the output file(s). When this option is active the output file names are based on the name of the memory areas in the `.scm` file.

23.12 More on linker rules

The linker rules file is a Scheme programming language source file, hence the `.scm` extension.

You do not need to be familiar with Scheme to specify linker rules, simply follow the examples to set things up.

The examples present a Scheme "program" with a single `memories` variable bound to a data structure. The linker reads the file with a Scheme interpreter, then examines the `memories` variable's contents. This has two implications: a) the file's syntax is Scheme-dictated, based on s-expressions[5]; and b) more elaborate programs using Scheme macros are possible. generate the memory rules.

23.13 Command line options

This section details `ln68k` command-line options.

23.13.1 Options overview

If the linker is run without command-line arguments, it will indicate that object files are required:

```
$ ln68k
the linker requires some object files to work with (use --help for help)
Terminating due to errors
```

A more detailed help message can be requested with the `--help` option:

```
$ ln68k --help
Calypsi linker for Motorola 68000 version 5.18

Usage: ln68k [--version] [-o|--output-file OUTPUT-FILE] [-g|--debug]
             [--semi-hosted] [--no-data-init-table-section]
             [--override IDENTIFIER] ([--cross-reference] |
             [--no-cross-reference]) [--rtattr NAME=VALUE] [--verbose] [-l]
             [--list-file LIST-FILE] [--memories-expression EXPRESSION]
             [--program-root SYMBOL] [--root-symbol SYMBOL]
             [--program-start SYMBOL] [--copy-initialize SECTION]
             [--no-copy-initialize SECTION] [--no-automatic-placement-rules]
             [--raw-multiple-memories] [--no-merge-raw-memories]
             [--force-output] [--no-auto-libraries] [--cstartup VALUE]
             [--no-tree-shaking] [--output-format FORMAT] [--sstack-size SIZE]
             [--stack-size SIZE] [--heap-size SIZE] ([--hosted] | [--rom-code])
             [--core CORE] [--target TARGET] [--hunk-max-size SIZE] [FILE...]
  use 'ln68k --help' for detailed help

Available options:
  --version               Display version number
  -o,--output-file OUTPUT-FILE
                          Name of output file
  -g,--debug              Produce debugging information
  --semi-hosted           Enable debug stubs for semi-hosting
  --no-data-init-table-section
                          Do not generate any data_init_table section (mainly
```

(continues on next page)

[5] https://www.wikipedia.org/wiki/S-expression

(continued from previous page)

```
                        useful for assembly projects)
--override IDENTIFIER   Override symbol in archive library (treat it as weak)
--cross-reference       Include cross reference and map information in the
                        list file (this is the default)
--no-cross-reference    Do not include cross reference and map information in
                        the list file
--rtattr NAME=VALUE     Specify runtime attribute to select specific
                        alternative from library (e.g. printf=float,
                        scanf=nofloat)
--verbose               Generate more detailed output
-l                      Generate a list file, defaults to 'ln68k.lst'
--list-file LIST-FILE   Generate list file, using given name
--memories-expression EXPRESSION
                        Expression that extracts the list of memory
                        descriptions from the .scm file, defaults to
                        'memories'
--program-root SYMBOL   Program root point, defaults to
                        '__program_root_section'
--root-symbol SYMBOL    Add a root symbol
--program-start SYMBOL  Program start symbol, defaults to '__program_start'
--copy-initialize SECTION
                        Override default and initialize this section by
                        copying
--no-copy-initialize SECTION
                        Override default and do not initialize this section
                        by copying
--no-automatic-placement-rules
                        Do not add rules to a .scm linker rules file
--raw-multiple-memories Allow multiple memories for raw style output format
--no-merge-raw-memories Never attempt to merge raw memories
--force-output          Ignore (some) errors and attempt to generate output
--no-auto-libraries     Do not automatically add runtime libraries
--cstartup VALUE        Define the C startup to be used, synonym to '--rtattr
                        cstartup=VALUE'
--no-tree-shaking       Do not perform tree shaking and keep unused section
                        fragment in output
--output-format FORMAT  Format, one of 'intel-hex', 'S-record', 'S19', 'S28',
                        'S37', 'raw', 'hunk', 'tos' or 'pgz' (in addition to
                        the ELF/DWARF output)
--sstack-size SIZE      Supervisor stack size override (overrides size
                        defined in the .scm file)
--stack-size SIZE       Stack size override (overrides size defined in the
                        .scm file)
--heap-size SIZE        Heap size override (size normally defined in the .scm
                        file)
--hosted                Initialize data sections in place by loading
                        executable (hosted enviroment)
--rom-code              Run from ROM/Flash, initialize data sections by
                        copying from ROM
--core CORE             Core, one of '68000', '68010', '68020', '68030',
                        '68040', '68060' or '68080' (defaults to '68000')
--target TARGET         Target system, one of 'Amiga', 'A2560U', 'A2560K' or
                        'TOS' (defaults to embedded/ROM use, if omitted)
--hunk-max-size SIZE    Largest Hunk size allowed, defaults to 256K (or
                        largest single object in the application)
-h,--help               Show this help text
```

23.13.2 Options in detail

`--output-file, -o`

Use this option to specify the name of the output executable file. If not given, the output file is `aout` with a file extension based on the format of the file, e.g. `aout.elf` or `aout.hex`.

`--debug`

Merge DWARF symbolic debugging information from the object file and output that in ELF executable file.

`-g`

Synonym for `--debug`.

`--semi-hosted`

Enable debug stubs for semi-hosting support. The program is linked with stubs for semi-hosting in the debugger. This allows the debugger to support standard I/O on behalf of the target and various other low level operations, such as capture `exit()` and implement `assert()`.

`--no-data-init-table-section`

Do not create any data initialization sections. This is mainly intended for assembly projects, but it can also be used for C if you do not want any initialization of static data objects to be done.

`--override`

Treat specified symbol as weak when taken from a library. This can be used if you want to replace a specific routine in a library with your own variant.

`--cross-reference`

Also include cross reference information in the list file. This is enabled by default.

`--no-cross-reference`

Do not include cross reference information in the list file.

`--rtattr NAME=VALUE`

Used to control selection of a specific variant of a module that exists in multiple weak versions. A typical situation is to select a custom C startup module or other variant of `exit()`.

It can also be used to override the automatic selection of `printf()` and `scanf()` formatter capability.

`--cstartup=VALUE`

Select a specific custom C startup module to be used. This is a slightly easier way than using the `--rtattr` option.

`--verbose`

Ask the linker to be more talkative. Mainly useful in cases where you run into errors. This option will generate a lot more information about the situation.

`-l`

Generate a list file. The name used is the name of the input file (ignoring any directory path) with a `.lst` file extension. See also `--list-file`.

`--list-file`

Generate a list file. The name of the list file is given as argument to this option. See also `-l` to generate a list file based on the source filename.

`--memories-expression`

This is the expression used to extract the memory description after reading a `.scm` rules file. This is set to `memories` by default and that should suffice in almost every situation.

`--program-root`

This is the root of the application and everything referenced from this section, directly or indirectly is part of the application.

If you are building a normal C project you should not change this.

`--root-symbol`

Add a root symbol that is regarded as something to always include in the build. This can be used to override a specific symbol, or if you have a root inside a library as they are otherwise ignored.

`--program-start`

This symbol defines the first actual executable instruction in the program. This is used as the entry point which is defined in some output formats.

If you are building a normal C project you should not change this.

`--output-format`

With this option you specify an additional file format for the produced executable application. An ELF executable is always produced.

`--raw-multiple-memories`

This option enables output to multiple files in the Raw file format. By default only one area is allowed and an error will result if the application uses more than one program area. Specifying this option will also result in different output filenames that are based on the memory areas in the `.scm` file.

`--no-merge-raw-memories`

This option controls whether adjacent memories are merged in the RAW output format.

`--heap-size`

The heap size can be defined in the linker control file (`.scm`). This option makes it possible to override the heap size and is useful if you use a common linker control file and want to avoid modifying the `.scm` file.

Note: The heap is completely removed if you set the size to `0`. If the application tries to use the heap in that situation, the linker will complain that the heap section is undefined.

`--sstack-size`

The supervisor stack size is normally defined in the linker control file (`.scm`). This option makes it possible to override the supervisor stack size and is useful if you use a common linker control file and want to avoid modifying the file.

`--stack-size`

The stack size can be defined in the linker control file (`.scm`). This option makes it possible to override the stack size and is useful if you use a common linker control file and want to avoid modifying the file.

`--hosted`

Specifies a hosted environment. Data object initialization by copying from ROM is omitted; instead, the application is loaded into memory, and initialized data objects receive values during the load action.

`--rom-code`

This is the opposite of `--hosted`. Data objects gets initialized by copying from ROM. This is the default and allows the application to be written to ROM or flash and started by turning the device on.

`--copy-initialize`

This option takes a data section name as argument. You can use this option to override the default of whether the section is initializated by copying or by loading.

This option can be used together with the `--hosted` option to make an exception for a given data section and have that initialized by copying. It can be used when the load mechanism is unable to load directly into a memory area. You can specify this option multiple times.

`--no-copy-initialize`

This option takes a data section name as argument. It is used to override the default of whether the section is initializated by copying or by loading. This option mostly exists for symmetry reasons. You can specify this option multiple times.

`--core`

Specifies the core used, currently 68000, 68010, 68020, 68030, 68040, 68060 and 68080 are supported. This is provided mainly for completeness and symmetry with the other tools. It has no practical meaning to the linker at the moment.

`--target`

Specifies a certain target system. Using this option may affect the setting of `--hosted` versus `--rom-code` and is preferred if the target system used is supported by this option.

`--no-automatic-placement-rules`

Do not attempt to add any rules, like section placments or missing blocks to a linker rules file. This is useful if you want to state everything explicitly, as it will result in errors if the linker rules (`.scm`) file is incomplete. In most cases you want to allow the linker to complete a partial linker rules file and this option should not be specified in such case.

`--force-output`

Ignore certain errors and allow output to be generated anyway. This is mainly intended for getting a list file to help understanding the cause of errors for troubleshooting.

`--no-auto-libraries`

The linker will attempt to pick a suitable C library based on the input. This command-line option disables this mechanism. If given you will need to specify the C library explicitly on the command line.

CHAPTER 24

Librarian

The `nlib` tool builds libraries (also called archives), which are single files containing multiple object files and a symbol index.

24.1 Creating a library

To create a library, list all object files (`.o` extension) and a single output file (`.a` extension).

```
$ nlib lib.a obj1.o obj2.o ashfx.o xtoal.o
```

The created library is typically used as input to the `ln68k` by specifying it on the command line. When linking with a library, only used objects are included in the final output.

24.2 File format

The file format is identical to `ar` on UNIX System V and GNU. The archive includes an index for faster object selection.

24.3 Command line options

Only `--version` and `--help` command-line options are supported.

To display the version of the linker, use `--version`:

```
$ nlib --version
Calypsi librarian for none version 5.18
```

CHAPTER 25

Debugger introduction

The Calypsi tool chain debugger is a source code debugger that enables you to observe program execution. It offers both interactive and script-based control. Modeled after command-line debuggers like GDB and LLDB, it shares many similarities, making it familiar to users of those tools.

25.1 Command line debugger

A command-line debugger provides powerful commands that are easy to remember and understand. You can use descriptive commands like `step`, `break` and `continue` rather than a set of bindings to function keys and other more or less cryptic key bindings. Thanks to short forms and repeating the last command with a return key press, you can execute common and repetitive actions with single or few keystrokes.

The debugger also supports various front-ends like Visual Studio Code, Eclipse, and Emacs, leveraging the GNU project's MI (Machine Interface) protocol for this integration.

The debugger also provides powerful scripting abilities via the Lua language. You can use Lua to configure debugging sessions, create your own commands and even make your own debugger plug-ins. The debugger offers powerful scripting via Lua. You can use Lua to configure debugging sessions, create custom commands, and develop debugger plug-ins. This is possible because the complete debugger functional API is exposed to Lua. The Lua API further extends capabilities by allowing definition of new console commands and subscription to internal debugger event notifications.

Thus, with the Calypsi debugger, you gain the advantages of both the command-line environment and integrated graphical development environments.

25.2 Running the debugger

The debugger's interface is command-line based, allowing use from a terminal or connection to an IDE.

25.2.1 Basic invocation

The debugger is started in the following way:

```
$ db68k [options] [debug-file] [options]
```

As everything on the command line is optional you can start the debugger without arguments. This will put you in an interactive debugger session:

```
$ db68k
```

Typically, specify the application program to debug on the command line:

```
$ db68k application.elf
```

Command line options and the application to debug can appear in any order on the command line.

Command line options are optional arguments that tune debugger behavior, always starting with a dash. Two variants exist: single- letter options with one dash, and long descriptive options with two.

To display the version of the debugger, use `--version`:

```
$ db68k --version
Calypsi debugger for Motorola 68000 version 5.18
```

25.3 Command line options

This section covers the `db68k` command-line options in detail.

25.3.1 Options overview

Use the `--help` option to display the available command-line options.

```
$ db68k --help
Calypsi debugger for Motorola 68000 version 5.18

Usage: db68k [--version] [-i|--interpreter INTERPRETER] [--nh] [--nx]
             [-e|--eval-command COMMAND] [-t|--tty TTY] [--batch]
             [--logging-enable] [--logging-to-file PATH] [--logging-overwrite]
             [--silent] [--extra-memory HEX-RANGE] [--stop-on-program-run]
             [--terminate-on-program-exit] [--exit-breakpoint] [--core CORE]
             [--target TARGET] [--supervisor-run] [--program-start SYMBOL]
             [--semi-hosted-root PATH] [--target-remote DEVICE]
             [--serial-speed SPEED] [FILE]
  use 'db68k --help' for detailed help

Available options:
  --version               Display version number
```

(continues on next page)

(continued from previous page)

```
-i,--interpreter INTERPRETER
                        Select command interpreter, one of 'Console', 'MI' or
                        'MI2' (defaults to 'Console')
--nh                    Do not read '~/.db68k'
--nx                    Do not read any '.db68k' file in any directory
-e,--eval-command COMMAND
                        Execute a single command (can be specified many
                        times)
-t,--tty TTY            Use TTY for input/output by the program being
                        debugged
--batch                 Exit after processing options
--logging-enable        Enable logging from start
--logging-to-file PATH  Enable logging from start to specified file
--logging-overwrite     Enable logging from start and overwrite the file
--silent                Generate less messages
--extra-memory HEX-RANGE Additional memory ranges to create (simulator use)
--stop-on-program-run   Stop and take control immediately when program is
                        started
--terminate-on-program-exit
                        Terminate execution of debugger if debuggee exits
                        (implies --exit-breakpoint)
--exit-breakpoint       Insert a breakpoint at 'exit()' to detect program
                        termination
--core CORE             Core, one of '68000', '68010', '68020', '68030',
                        '68040', '68060' or '68080' (defaults to '68000')
--target TARGET         Target system, one of 'Amiga', 'A2560U', 'A2560K' or
                        'TOS' (defaults to embedded/ROM use, if omitted)
--supervisor-run        Run program in supervisor mode (on-target debugging
                        only)
--program-start SYMBOL  Program start symbol, defaults to '__program_start'
--semi-hosted-root PATH Root directory of the semi-hosted file system
--target-remote DEVICE  Use remote device to communicate with target
--serial-speed SPEED    Set serial speed, one of 9600, 19200, 38400, 57600 or
                        115200 (defaults to 115200)
-h,--help               Show this help text
```

25.3.2 Options in detail

`--interpreter, -i`

Use this option to specify the interpreter to use. The debugger can execute either with a console interpreter or a machine interface (MI) interpreter. The console interpreter is suitable for interactive debugger sessions from the command line, which is the default. The machine interface interpreter is used when the debugger is controlled by an IDE.

`--eval-command, -e`

This option specifies a command to execute before the interactive session gains control. It can be used multiple times to specify multiple commands.

`--batch`

Terminate the debugger session after all `--eval-command` commands have been processed. This is useful for automated debugger execution from scripts.

`--tty, -t`

Specifies the application's console `tty`. If the application writes to or reads from the standard stream, this provides the `tty` device for that operation.

If not specified, console output from the application is written to the console interaction. In MI mode such console output is marked so that the IDE can see that it is stream output from the application.

`--semi-hosted-root`

Specifies the host-side directory serving as the target's semi-hosted file system. Defaults to the current directory. See Semi-hosting for details.

`--logging-enable`

Enables logging at debugger startup.

`--logging-to-file`

Specifies the logging output file.

`--logging-overwrite`

Specifies whether logging to the file should append or overwrite previously existing log file content.

`--extra-memory`

Specifies additional memory areas not part in the application that exists on a simulated target.

Memories in the simulator are normally created based on the memories found in the debug application image. This option allows additional memory regions to be created.

`--stop-on-program-run`

When the program starts, immediately take control and stop execution. This enables debugging from the very beginning (before `main()` is reached) without needing to set a breakpoint.

`--terminate-on-program-exit`

If the application exits, the debugger session also terminates. The default behavior is for the debugger to notify of the exit and remain active.

`--exit-breakpoint`

Insert a breakpoint at the `exit()` function. This option is not needed if you link with debug stubs, as exit handling is provided.

`--target-remote`

Specifies the communications device to connect to a target for debugging. This disables the built-in simulator and configures the debugger for remote debugging via the gdbserver protocol, typically used for debugging on real hardware.

`--serial-speed`

Set the speed of the target remote communications serial port defined with `--target-remote`. The speed defaults to 115200.

`--program-start`

This symbol defines the program's first executable instruction. For a standard C project, avoid changing this setting.

`--core`

Specifies the core used.

`--target`

Specifies a certain target system.

`--supervisor-run`

This option is relevant when connecting to a remote target system and tells the remote system that the application must be started in supervisor mode with interrupts disabled. The application will by default start in normal mode with interrupts enabled to mimic that the application has been loaded by some operating system.

This option is ignored by the simulator which starts the application as if a reset occurred.

CHAPTER 26

Console command line

The console interpreter provides an interactive command line with powerful features, including command completion, history, and editing.

Your command-line history is also saved between sessions in `~/.db68k/db68k.history`.

26.1 Command structure

Commands consist of one or more space-separated words. If a word is a unique match, it does not need to be fully spelled out.

An argument specific to the command (e.g., a string, file path, or boolean value) may follow the command.

26.1.1 Command completion

Command completion is available via the `TAB` key. Pressing `TAB` directly at the prompt lists all words that can start a command:

```
(db68k) <TAB>
b                enable           n                s
break            exec-file        next             set
c                file             nexti            show
cd               frame            ni               si
clear            ignore           p                source
complete         info             platform         step
condition        interpreter-exec plug-in          stepi
continue         interrupt        print            thread
delete           kill             pwd              tty
directory        lua              quit
disable          maintenance      r
disassemble      module           run
(db68k)
```

Pressing TAB completes the command to the left of the cursor as far as possible. If the word can be partially completed, the first TAB fills in as much as possible:

```
(db68k) int<TAB>
```

Results in:

```
(db68k) inter
```

Pressing TAB a second time results in:

```
(db68k) inter<TAB>
interpreter-exec  interrupt
(db68k) inter
```

26.2 A simple debugging session

To get a feeling for how to run a very simple debugging session here is how it can look.

```
$ db68k program.elf
Calypsi debugger for 68000
(db68k) b main
breakpoint 1
(db68k) info breakpoints
Num      Type            Disp  Enb  Lua/plug-in   Address     What
1        breakpoint      keep  y    no            0x80f1      main
(db68k)
```

Note: A command requires enough characters to be unique. For `info breakpoints`, `i b` suffices, as the interpreter considers all words for multi-word commands. The completer operates left of the cursor: `i<TAB>` shows all commands starting with 'i'. To complete to `info`, type `inf<TAB>` followed by `b<TAB>`. However, `i b<TAB>` expands directly to `i breakpoints` because no other command matches this two-word prefix.

To start debugging, use the `run` command (abbreviated to `r`), which begins execution from the start. The program will stop almost immediately upon hitting the breakpoint at the beginning of the `main()` function.

```
(db68k) r
running
   12 int const cglob_int = 12;
   13
   14 size_t fossasas;
   15
-> 16 int main () {
   17   basic_expr(glob_int + 21, 2, 3);
   18   int xx = fact(4);
   19   int b = vla1(4);
   20   int c = struct1(2);
0x80f1 in main() at main.c:16
(db68k)
```

The breakpoint is hit at address `0x80f1` in `main.c:16`. Source lines are shown as context, with the current line indicated.

You can now perform a couple of single steps:

```
(db68k) s
   13
   14 size_t fossasas;
   15
   16 int main () {
-> 17   basic_expr(glob_int + 21, 2, 3);
   18   int xx = fact(4);
   19   int b = vla1(4);
   20   int c = struct1(2);
   21   int d = struct11(2);
0x80f8 in main() at main.c:17
(db68k)
   10 int bar (long* p) {
   11   return *p + 2;
   12 }
   13
-> 14 long basic_expr (long a, long a2, int a3) {
   15   char cglob_int = 0; // block a variable in global scope
   16   if (a) {
   17     int b = 10;
   18     a = foo(b + cglob_int);
0x8000 in basic_expr() at basic_expr.c:14
(db68k)
```

The `step` command (abbreviated as `s`) steps one source line. Subsequent steps can be performed by simply pressing `RETURN`, which repeats the most recent command.

Each step advances us one line, updating the source context shown.

The step command steps into functions, as demonstrated by entering `basic_expr()`.

You can inspect variables with `info locals`.

```
(db68k) info locals
(int) a3 = 3
(long) a2 = 2
(long) a = 21
(db68k)
```

To trace your current position, use the `backtrace` (or `bt`) command.

```
(db68k) backtrace
#0 0x8000 in basic_expr() at basic_expr.c:14
#1 0x8137 in main() at main.c:17
(db68k)
```

To stop at a specific line further down the source code, use:

```
(db68k) b basic_expr.c:18
breakpoint 2
(db68k) c
running
program hit breakpoint 2
   14 long basic_expr (long a, long a2, int a3) {
   15   char cglob_int = 0; // block a variable in global scope
   16   if (a) {
   17     int b = 10;
-> 18     a = foo(b + cglob_int);
```

(continues on next page)

(continued from previous page)

```
   19     a += bar(&a2);  // take address of a2, forcing it to stack.
   20   }
   21   if (a > 2) {
   22     // Make the variable in global scope visible
0x8030 in basic_expr() at basic_expr.c:18
(db68k)
```

The `continue` command (or `c`) resumes execution (unlike `run`, which starts from the beginning).

Lets take a look at the local variables again.

```
(db68k) info locals
(int) b = 10
(char) cglob_int = 0
(int) a3 = 3
(long) a2 = 2
(long) a - value not available, reason: "a" has no valid value here
(db68k)
```

The local context now includes two additional variables: `b` and `cglob_int`. At this point, `a` has no value because its last use was on line 16, and it will receive a new value on the current line. Thus, the debugger knows about `a` but it currently holds no examinable value.

Step over two function calls and inspect the value of `a` using the `print` command, which can be entered as `p`:

```
(db68k) n
   15   char cglob_int = 0; // block a variable in global scope
   16   if (a) {
   17     int b = 10;
   18     a = foo(b + cglob_int);
-> 19     a += bar(&a2);  // take address of a2, forcing it to stack.
   20   }
   21   if (a > 2) {
   22     // Make the variable in global scope visible
   23     extern int const cglob_int;
0x8050 in basic_expr() at basic_expr.c:19
(db68k)
   17     int b = 10;
   18     a = foo(b + cglob_int);
   19     a += bar(&a2);  // take address of a2, forcing it to stack.
   20   }
-> 21   if (a > 2) {
   22     // Make the variable in global scope visible
   23     extern int const cglob_int;
   24     a = foo(cglob_int);
   25   }
0x8082 in basic_expr() at basic_expr.c:21
(db68k) p a
(long) $8 = 8
(db68k)
```

The variable `a` now holds a value from the function calls.

The local `char cglob_int` shadows a global variable of the same name. Stepping to line 24 makes the global `cglob_int` visible:

```
(db68k) n
   20   }
   21   if (a > 2) {
   22     // Make the variable in global scope visible
   23     extern int const cglob_int;
-> 24     a = foo(cglob_int);
   25   }
   26   return a + a3;   // a3 alive and on the stack
   27 }
   28
0x8098 in basic_expr() at basic_expr.c:24
(db68k) p cglob_int
(int const) $9 = 12
(db68k)
```

The global variable is now visible again, with a different value and type.

If you no longer need the first breakpoint, you can disable it using the `disable` command, rather than deleting it:

```
(db68k) disable 1
(db68k) info breakpoints
Num      Type            Disp  Enb  Lua/plug-in   Address     What
1        breakpoint      keep  n    no            0x80f1      main
         breakpoint already hit 1 time
2        breakpoint      keep  y    no            0x8030      basic_expr.c:18
         breakpoint already hit 1 time
(db68k)
```

The first breakpoint still exists but is not enabled (`n` in the `Enb` column). The debugger also tracks breakpoint statistics.

CHAPTER 27

Remote debugging

This chapter describes how to set up remote debugging to a target. In this scenario, the debugger runs on the host and connects to, controlling, an application on actual hardware.

The gdbserver protocol is used for communication. It can be implemented by a small agent on the target or a bridge process on the host that translates between gdbserver and hardware-specific protocols.

27.1 Serial port

Target debug support can be implemented by placing a small debug agent software on the target. The RS-232 serial port can be used for this. Since most host computers lack these ports, a USB-to-serial converter (e.g., FTDI USB-RS232, often with bare wires requiring a soldered connector) can be used.

Most major operating systems include drivers for the FTDI USB-RS232 cable. If you choose other cables, you may need to install a driver, which you can obtain from the USB-RS232 cable vendor.

27.1.1 Finding the device

On Linux and macOS, the serial port is typically found in `/dev/ttyUSBn`, `/dev/ttySn`, or similar. Use the `dmesg` command to query for USB devices:

```
$ sudo dmesg | grep USB
...
[109191.143623] ftdi_sio 1-1:1.0: FTDI USB Serial Device converter detected
[109191.151619] usb 1-1: FTDI USB Serial Device converter now attached to ttyUSB1
```

Here you can see that a FTDI serial device is attached to `/dev/ttyUSB1`.

Note: If you solder a serial port connector, be careful about which pin numbering. Also note that a DB-25 connector has the receive and transmit pins (numbers 2 and 3) are wired oppositely compared to a DE-9 connector.

27.1.2 Setting the speed

The speed can be set with the `--serial-speed` command-line option, which defaults to 115200 if not specified.

27.1.3 Testing the port

If you have the `screen` utility program installed it can be started with parameters to test that the serial port works:

```
$ screen /dev/ttyUSB1 115200
```

Once started, you can verify debugger agent responsiveness by pressing any character; a `$S13#b7` packet reply confirms it.

You can exit by pressing Control-A followed by a backslash (`c-a \`). Some screen versions require Control-A followed by Control-backslash.

27.1.4 Installing the agent

The Calypsi remote debugger agent for the serial port must be downloaded, built, and executed on the target hardware. Download it from https://github.com/hth313/Calypsi-remote-debug.

27.1.5 Starting the debugger

Once the debugger agent runs on the hardware and the communications channel is configured, you can start the Calypsi `db68k` using:

```
$ db68k --target-remote /dev/ttyUSB1 application.elf
```

Note: The application will start in normal mode (also called user mode) with interrupts enabled. This mimics a situation where the application has been loaded by an operating system. You can also start the application in supervisor mode with the interrupt mask set to 7 using the `--supervisor-run` command-line option.

CHAPTER 28

Visual Studio Code

Visual Studio Code[6] is a lightweight, yet powerful source code editor that runs on multiple platforms.

With the C/C++ for Visual Studio Code extension[7] it is possible to use the Calypsi C compiler tool chain in Visual Studio Code.

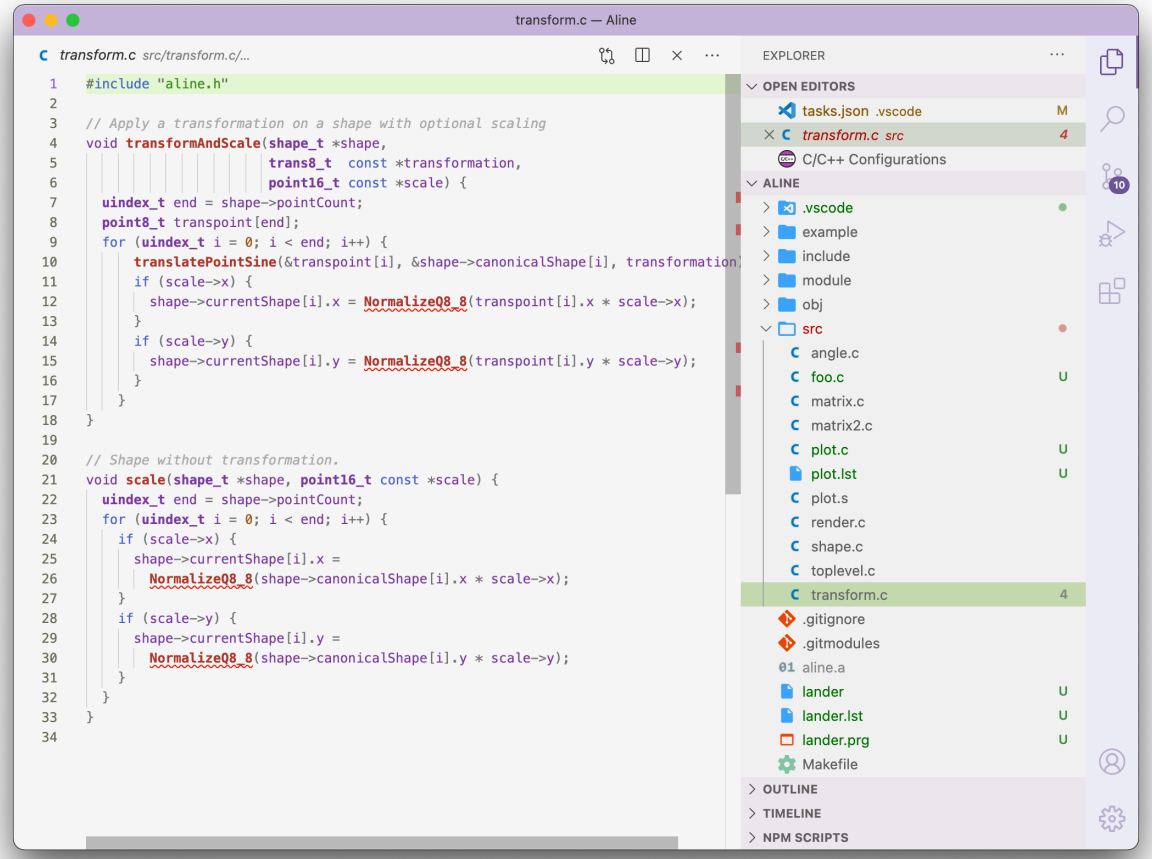
Visual Studio Code can be tailored to use various build and debugger tools. Its debugger integration, based on MI (Machine Interface), is supported by the Calypsi `db68k` debugger.

Visual Studio Code is an excellent GUI choice for Calypsi, being actively developed, lightweight, responsive, and highly configurable.

This guide does not fully document Visual Studio Code. For detailed information, refer to the official documentation[8].

Some basics on getting started are covered here, as its operation may differ from traditional IDEs.

[6] https://code.visualstudio.com/
[7] https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools/
[8] https://code.visualstudio.com/docs



28.1 Installation

Installation is easy to do by downloading it from the official site. After that you can update from inside Visual Studio Code itself.

Note: If you use Arch Linux or a derivative like Manjaro, install VS Code via its package system. The community `code` package currently cannot install the vscode-cpptools extension due to licensing. To make it work, install the Microsoft-branded `visual-studio-code-bin` from the AUR.

28.2 Project setup

In contrast to many IDEs you will not find any project setup in the menus.

To create a new project from scratch, use `File>Open` and create a new folder for your project and then press `Open`. You now have an empty project directory.

For an existing project, select `File>Open`, browse to its top folder, and press `Open` to load.

Project-specific files are stored in a `.vscode` directory within the project. Initially, this directory is empty.

28.3 Building

Once the project is populated and a `Makefile` created, add a build task. Select `Tasks>Configure Tasks...`, then `Create tasks.json file from template`, and `Others`. Edit the resulting `tasks.json` file, which may resemble:

```
{
    "version": "0.2.0",
    "tasks": [
        {
            "label": "build",
            "type": "shell",
            "command": "make -k",
            "options": {
                "cwd": "${workspaceRoot}"
            },
            "problemMatcher": [
                "$gcc"
            ]
        },
        {
        "label": "clean",
        "type": "shell",
        "command": "make -k clean",
        "problemMatcher": [],
        }
    ]
}
```

28.4 Running the debugger

The `db68k` debugger can be used inside Visual Studio Code. For basic use it uses a 68000 simulator and a memory system that is configured based on the executable image.

28.4.1 Configuration

Before starting the debugger, add a launch configuration.

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "C++ Launch",
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceRoot}/src/myproject",
            "args": [],
            "stopAtEntry": false,
            "cwd": "${workspaceRoot}/src",
            "environment": [],
            "externalConsole": false,
            "linux": {
                "MIMode": "lldb"
                "miDebuggerPath": "/usr/local/bin/db68k",
                "launchCompleteCommand": "exec-run",
                "setupCommands": []
            },
            "osx": {
                "MIMode": "lldb",
                "miDebuggerPath": "/usr/local/bin/db68k",
                "launchCompleteCommand": "exec-run",
                "setupCommands": []
            },
            "windows": {
                "MIMode": "lldb"
            }
        },
    ]
}
```

There are a couple of things to specify here.

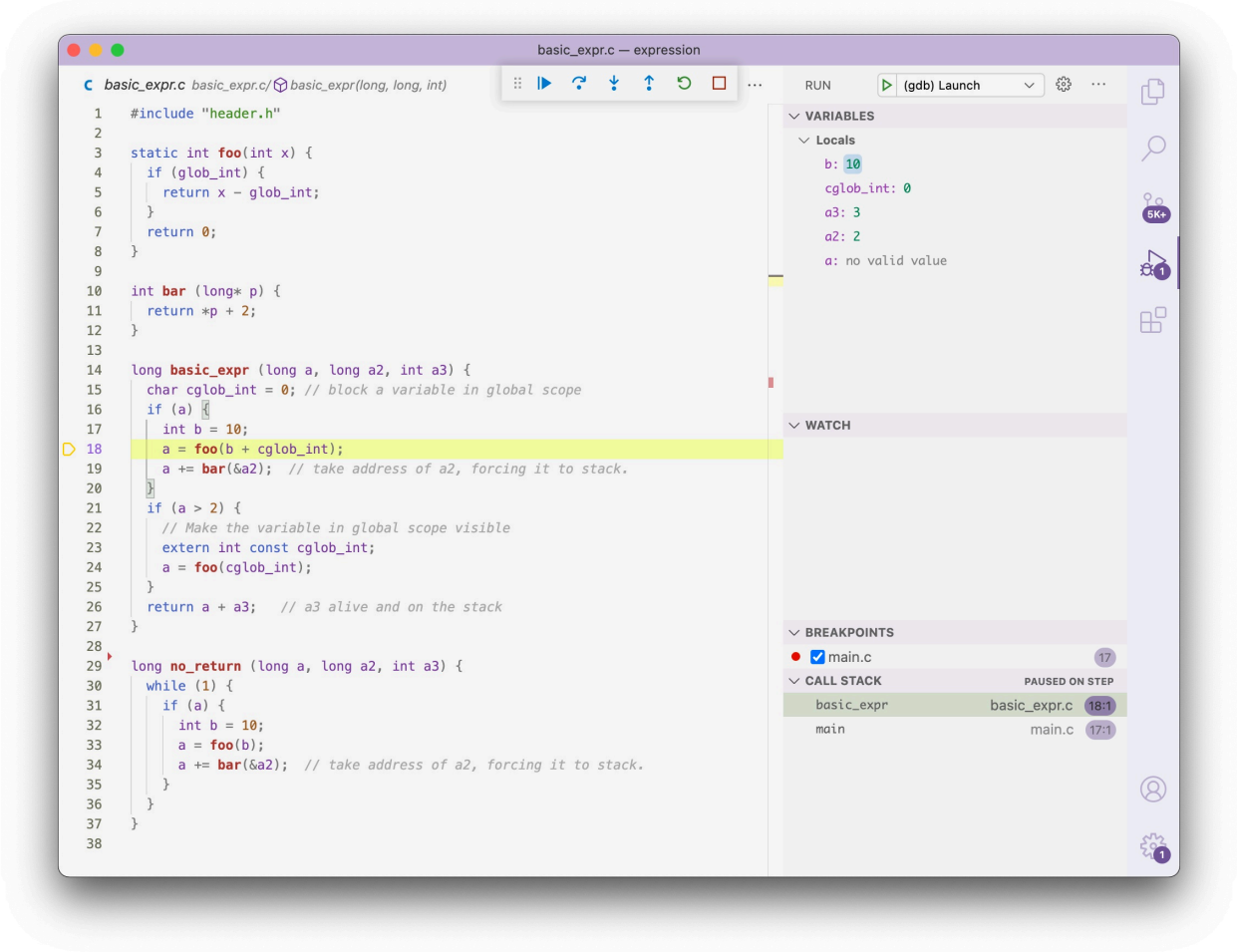
1. You need to specify that you are using `/usr/local/bin/db68k` as the debugger using `miDebuggerPath`
2. You also need to tell that you want to use `lldb` as `MIMode`[9]

The launch configuration is a text file stored in `.vscode/launch.json`.

[9] Even though `db68k` is much closer to `gdb` than `lldb` at the command level, you need to specify `lldb` as the MI mode. The reason is that in the `gdb` mode, Visual Studio Code works around a bug on UNIX style platforms for the MI command `-exec-interrupt` (which interrupts execution). The work around is to send a signal to a real UNIX process to interrupt it. This does not work with db68k as the application being debugged is not a UNIX process at all. Instead you need to use the `lldb` mode, where the MI command `-exec-interrupt` works properly, just as it does in `db68k`.
Reference https://sourceware.org/bugzilla/show_bug.cgi?id=20035

28.4.2 Running the application

Once configured, run your application using `Run >> Start Debugging` (`F5`). Consider inserting a breakpoint or interrupting execution once started. If debugger windows (variables, call stack, etc.) don't appear, click the bug icon to switch to the debugger view.



CHAPTER 29

Lua scripting

The debugger can be controlled and extended using the Lua language[10]. Scripting allows you to write programs that execute inside the debugger, from simple batch-like scripts to advanced plug-ins that add custom commands, listen to internal events, and control execution.

From a high-level perspective, Lua and the debugger enable various tasks:

- Perform setup tasks to prepare your debug session
- Simplify repetitive tasks
- Run fully automated tests
- Extend functionality; for instance, to introspect the state and behavior of an operating system or protocol stack. These commands integrate into the existing set and share a command-line interface.

You can choose any automation level, from running the entire session under script control to providing high-level functionality with commands.

29.1 Running a script

To get started, this simple Lua program just displays the registers:

```
local db = require('db')

print(db.consoleCommand('info registers'))
```

If saved in `script-name.lua` (in the current directory), you can run the script with:

```
(db) source script-name.lua
```

Debugger functionality is provided in the `db` table which is created before the script is started. To access it, you need to use `require`. Here we store the debugger table in the local variable `db`.

[10] https://www.lua.org/

Note: Use the `--eval-command` (can also be used as `-e`) to invoke a Lua script from the command line and to make it execute automatically when the debugger starts. The `--eval-command` works with any command and can be specified multiple times on the command line.

29.2 Lua environments

Each time you run a Lua script, a new fresh Lua state is created. This means you will start out with the standard Lua and debugger libraries available imported, but not anything else. If you create a global variable, it will not normally be present the next time you run the script.

However, if a script provides new debugger functions (e.g., console commands), the Lua state is saved with the command. Subsequent invocations of that command reuse the same state, retaining global variables and closures. This allows commands to operate within their established state and depend on prior actions within that state, including previous uses of the same or other commands created by the script.

A Lua plug-in typically registers functions with the debugger for specific circumstances. These functions always execute in the state in which they were installed.

It is good practice to provide related functionality (e.g., a command set and plug-in behavior) in its own script. Multiple plug-ins can be installed via source invocation, allowing each to have its private shared state while co-existing separately.

29.3 Command interpreters

The debugger provides two sets of command interpreters, console commands and MI commands.

Console commands are intended for humans, provide command completion when being entered and formatted output that is easy to read.

The Machine Interface (or MI for short), was introduced by GDB to provide a more precise and stable command protocol, intended for (graphical) debugger front-ends. The GDB MI command set is supported by the debugger, allowing existing graphical debugger front-ends to be used.

Many commands can be accessed from either of these command sets, while some commands are only available in one of them.

The Lua interface allows both command sets to be used. In addition, adapted MI commands and additional functionality are also provided.

29.4 Console commands

The `db` table allows sending console commands to the debugger. These commands execute as if typed at the terminal, and their string results are passed back to Lua.

If you need to process the result beyond displaying or ignoring it, parsing is required. However, writing such parsers quickly becomes tedious, as the text output is primarily for display.

29.5 MI commands

MI commands offer precise interaction for debugger front-ends, making them well-suited for Lua scripts. Although commands are string-based, results are converted to nested Lua tables with basic types (booleans, numbers, strings).

This significantly simplifies interpreting results from Lua-to-debugger calls.

29.6 Adapted MI commands

Adapted MI commands are basically MI commands converted to be a more natural fit for Lua. This is done in two ways. The actual commands take arguments like any ordinary function in Lua. A return values is the value returned or a failure, instead of the MI style result table hierarchy.

29.7 Choosing a command set

In most cases you will probably want to use the adapted MI command set as it provides the most natural Lua experience.

The console and MI command set may be useful if you have familiarity with either or prefer the alternative output form provided.

29.8 Arrays and Lua

Lua array indices start at 1, unlike C. This mismatch has implications when working with debugger C expressions from Lua.

To simplify working with C expressions, array results can start at 0 in Lua, mirroring C behavior. However, the ipairs iterator (designed for Lua's 1-based indexing) will skip the element at index 0.

Shifting the index in expressions is an alternative, but it would likely be more confusing and error-prone, especially given the context of C expressions.

29.9 Debugger API

`db:stepInstruction()`
: Single step once at instruction level. This function will quickly return an indication of whether it was successful in starting the target. Actual progress of execution will be posted as notifications

  Corresponding MI instruction is `-exec-step-instrucion`.

CHAPTER 30

Lua data types reference

This section describes the data structures used. Complex types (beyond booleans, numbers, or strings) are represented by tables.

30.1 Address

The debugger uses internal addresses, composed of a numeric value, memory type, and optional bank number. In Lua, addresses are represented by a table with these fields:

Table 30.1: Address type

field	type	description
formatted	string	The complete print value of the address location.
address	integer	The raw address bits of the location.
memory	string	The kind of the memory the address points to, can typically be “code” or “data” (string).
bank	integer	The bank number, or nil if this address location does not have any bank associated with it.

The `formatted` field is set when the value is returned to Lua. If you provide an address as an argument to a function call in the API, you do not need to provide a `formatted` field.

30.2 Adapted MI functions

Many Lua functions are adapted from machine interface (MI) commands, used for communication between a debugger front end (like VS Code) and the debugger.

Functions are available in the db table. The Lua interface translates arguments for these functions and passes them to the debugger command handler.

Results are passed back and translated to Lua. As the underlying mechanism is a command protocol, the result table is a bit more elaborate than what might be expected.

A top-level result has the following fields:

field	type	description
class	string	The result class (e.g., done for success, `error` for failures). A complete list is below.
value	integer	The result, which depends on the function called.
type	string	The reply type. For function calls, this is `result`.

CHAPTER 31

Lua function reference

This section covers functions provided by the debugger module, grouped by family for easy reference.

31.1 Execution

31.1.1 db:stepInstruction()

Steps once at the instruction level.

The result indicates only if the target was notified to start. An error result means the target is not in a state to start.

To monitor progress and result of execution, you need to listen to notifications.

31.2 Breakpoint

31.2.1 db:breakDelete(number(s))

Removes one or more breakpoints.

The argument is the breakpoint number(s) to remove (an integer or array of integers).

31.2.2 db:breakInsert(location, &callBack)

Inserts a new breakpoint.

The first argument, a Lua table, may contain breakpoint attributes:

Table 31.1: Breakpoint attributes

field	type	description
temporary	boolean	set a temporary (one-time) breakpoint
hardware	boolean	Use a hardware breakpoint.
pending	boolean	Allow pending breakpoint.
disabled	boolean	Start with the breakpoint disabled.
tracePoint	boolean	This is a tracepoint.
condition	expr	Set a conditional expression.
ignore	integer	Set initial ignore count.
thread	integer	Breakpoint is valid for a given thread.

The second argument describes the breakpoint's location, which can be:

Table 31.2: Breakpoint location

kind	type	
address	table	an address table
file and line	table	{ file : string, line : integer
function	string	function name or symbol

The third if present should be a Lua function. This function will be called as followed when the breakpoint is hit:

```
cont = callBack(object, breakpointNumber)
```

The callBack's first argument is the object table used during breakpoint creation; the second is the breakpoint number. If callBack returns boolean false, the breakpoint is not taken, and execution resumes silently, without user interface feedback.

The callback occurs in the execution thread and blocks execution. If you intend to continue, return as quickly as possible.

If execution stops, you can process it within the callback or detect the stop via notifications. Be aware that processing in the callback will prevent user feedback until it returns.

Index

Non-alphabetical

_NearBaseAddress
    symbol, 146

A

A2560, 34
A2560 Foenix U
    example, 9
address space
    syntax, 21
address spaces, 20, 21
.align
    directive, 122
aligned
    attribute, 59
alignment, 47, 56, 59
    of sections, 115
allocated memory, 147
Amiga, 33
    example, 8
    interrupt, 60
    NDK, 33
Amiga library
    auto open, 33
Arch Linux
    installation, 5
    uninstalling, 5
archive, 158
.ascii
    directive, 122
.asciz
    directive, 122
assembler, 113
    constants, 116
    directives, 118
    expressions, 115
    invocation, 129
    labels, 114
    local labels, 117
    numbers, 116
    operators, 116
    symbols, 114
    syntax, 113
assembly language, 113
assembly source
    generated by compiler, 38
assembly source file
    option, 43
attribute
    aligned, 59
    far, 59
    interrupt, 59
    intrinsic, 60
    near, 59
    saveds, 61
    section, 59
    task, 61
attributes, 57
auto open
    Amiga library, 33
auto variables, 19
auto_type, 64

B

back quoted symbols, 114
bare metal, 139
base address
    definition, 145
bit fields, 50
    volatile, 55
block
    section in linker, 144
bool type, 49
bss section, 81, 114
building
    C library, 98
built-in, 69
built-in functions, 69
.byte
    directive, 122
byte order
    macros, 76
.byte0
    relocation operator, 124

C

C language, 13, 26
C library, 88
    building, 98
    errno, 94
    replace function, 96
C startup, 96
    replacing, 96
calling convention, 107
cfar
    section, 23, 82, 84
character type, 50
code

section, 23, 82, 84
code model, 16
command
structure, 167
command completion, 167
comments
assembly, 113
compiler
dependency file, 45
diagnostics, 38
errors, 38
generate assembly source, 38
invocation, 35
list file, 36
options, 39
warnings, 38
conditional assembly, 114
const
qualifier, 55
constants
assembler, 116
constraints
inline assembler, 110
context
program, 168
conversions
implicit, 30
core selection
option, 47, 135
cross call
optimization, 102
option, 44
cross compiler, 13
cross reference, 147
customizing
system startup, 96

D

data initialization, 25
omitting, 154
data model, 16
far-only, 16
large, 16
small, 16
data pointer, 51
size, 51
data section, 81, 114
initialization, 82
data_init_table
section, 23, 82, 85
Debian
installation, 3
uninstalling, 4
debug library, 88
debugger
command interpreter, 163
logging, 164
starting, 161
debugging information
macros, 76
option, 43, 135
dependency file
compiler, 45
devices
supported, 13
diagnostics
compiler, 38
pragma directive, 68
directive
.align, 122
.ascii, 122
.asciz, 122
.byte, 122
.equ, 119, 120
.equlab, 120
.extern, 120
.fillto, 122
.global, 120
.globl, 120
include binary, 121
.long, 122
.macro, 123, 126
.public, 120
.quad, 122
runtime model, 121
.section, 114, 119
.space, 122
.word, 122
directives
assembler, 118
distribution
installation, 2
DWARF
output, 151

E

ELF
output, 151
embedded systems, 139
endian
macros, 76
enumeration type, 53
.equ directive, 119, 120
.equlab directive, 120
errno
C library, 94
error
pragma directive, 68
errors
compiler, 38
internal, 38
example
A2560 Foenix U, 9
Amiga, 8
Hello World, 6
exit, 95
expressions
assembler, 115
extended attributes, 57
.extern directive, 120

F

far
attribute, 59
section, 23, 82, 84
far-only
data model, 16
Fedora
installation, 4
uninstalling, 5
file extensions, 14
file inclusion, 114
file streams
initialization, 25

fill word, 143
.fillto
    directive, 122
floating point
    precision, 51
    setting, 16
    types, 16, 51
floating point types, 29
Foenix
    target, 47, 135
Foenix A2560, 34
Foenix A2560 U
    example, 9
formatter
    printf, 94
    scanf, 95
function pointer
    size, 51
function prototypes, 30
functions, 69

G

global variables, 20

H

headers
    search path, 36
heap
    initialization, 25
    specify size, 145
Hello World
    example, 6
host based systems. operating system, 139

I

IDE
    Visual Studio Code, 174
IEEE 754, 51
ifar
    section, 85
include binary
    directive, 121
include files, 114
include search
    option, 42, 134
include search path, 36
index type
    pointer, 51
inear
    section, 84
initialization
    of data, 25, 82
inline assembler, 109
    constraints, 110
    substitutions, 111
    volatile, 109
inlining
    optimization, 100
inlining functions
    not inlined, 102
    option, 44
installation, 2
    Arch Linux, 5
    Debian, 3
    Fedora, 4
    Linux, 35
    macOS, 3
    Manjaro, 5
    multiple, 6
    multiple versions, 6
    Ubuntu, 3
    Windows, 5
integer types, 29, 49
Intel hex
    output, 151
internal errors, 38
interrupt
    Amiga style, 60
    attribute, 59
    vector, 60
intrinsic, 69
    attribute, 60
intrinsic functions, 69
intrinsics, 17
invocation
    assembler, 129
    compiler, 35

K

keywords
    address spaces, 21

L

labels
    assembler, 114
    global, 120
    location counter, 116
    weak, 120
language extensions
    auto_type, 64
    overloaded functions, 63
    statement expression, 63
    typeof, 64
large
    data model, 16
librarian, 158
library
    building, 98
    files, 88
    replace function, 96
linker, 136
    cross reference, 147
    scatter, 146
linker rules, 24, 138, 143
Linux
    installation, 35
    uninstalling, 4, 5
list file
    compiler, 36
    option, 42, 134
list files
    linker, 147
load address
    option, 157
local labels
    assembler, 117
local labels inside macro, 126
local variables, 19
location counter, 116
.long
    directive, 122
Lua, 179
lua

adapted MI functions, 185
address type, 185
breakpoints, 187
execution control, 187

M

macOS
installation, 3
uninstalling, 3
.macro, 126
directive, 123
macro, 126
local label, 126
macro definition
option, 43, 134
macro definitions
showing, 44
macro language, 126
macros
byte order, 76
debugging information, 76
endian, 76
predefined, 75
version, 76
Manjaro
installation, 5
uninstalling, 5
memory, 142
address range, 142
fill, 143
name, 142
placement group, 146
size, 147
message
pragma directive, 68
Motorola S-record
output, 151

N

NDK
Amiga, 33
.near
relocation operator, 125
near
attribute, 59
base address, 145
section, 23, 82, 84
near code
section, 84
no-init
section, 82
numbers
assembler, 116

O

omit data initialization, 154
operator
.sectionEnd, 125
.sectionSize, 125
.sectionStart, 125
operators
assembler, 116
relocation, 123
section, 125
optimization
cross call, 102
inlining, 100
optimizer, 17
option, 44
tuning, 44
option
core selection, 47, 135
cross call, 44
debugging information, 43, 135
function inlining, 44
include search, 42, 134
list file, 42, 134
load address, 157
macro definition, 43, 134
optimizer level, 44
output file, 42, 134
runtime attribute, 45, 135
signed char, 44
size of double, 43
switch control, 45
system include search, 42, 134
target selection, 47, 135, 157
unsigned char, 44
vector sections, 45
version, 42, 134
weak symbols, 45, 135
option: assembly source file, 43
options
compiler, 39
output
DWARF, 151
ELF, 151
Intel hex, 151
Motorola S, 151
raw, 151
S19, 151
S28, 151
S37, 151
SREC, 151
output file
option, 42, 134
output format
ELF/DWARF, 151
Intel hex, 151
Motorola S-record, 151
raw, 151
overloaded functions, 63

P

placement control, 24
placement group
memory, 146
pointer
index type, 51
types, 51
pointers, 21
pragma directive
diagnostics, 68
error, 68
message, 68
require, 69
rtattr, 69
runtime attribute, 69
section, 68
warning, 68
precision
floating point, 51
predefined

macros, 75
preprocess, 44
preprocessor, 74
print formatter, 94
program context, 168
prototypes
function, 30
ptrdiff_t type, 52
.public directive, 120

Q

.quad
directive, 122
quoted symbols, 114

R

raw
output, 151
read only
data section, 114
read-only
section, 82
record type, 52
relocation operators, 123
relocations, 123
replace function
in C library, 96
require
pragma directive, 69
required symbols, 121
rodata section, 82
ROM based systems, 139
rtattr
pragma directive, 69
rules for linking, 138, 143
runtime attribute
option, 45, 135
pragma directive, 69
runtime library, 88
runtime model directive, 121

S

S19
output, 151
S28
output, 151
S37
output, 151
saveds
attribute, 61
scan formatter, 95
scatter
linker, 146
search path
headers, 36
.section
directive, 114
section, 23, 114
alignment, 115
attribute, 59
block, 144
bss, 81, 114
cfar, 23, 82, 84
code, 23, 82, 84
data, 81, 82, 114
data_init_table, 23, 82, 85
directive, 119
far, 23, 82, 84
fragments, 114, 147
ifar, 85
inear, 84
kinds, 114
modifiers, 115
near, 23, 82, 84
near code, 84
no-init, 82
operators, 125
placement, 24, 143
pragma directive, 68
read only data, 114
read-only, 82
stack, 144
switch, 23, 82, 84
text, 81, 114
types, 82
zfar, 23, 82, 84
znear, 23, 82, 84
.section directive, 119
.sectionEnd
operator, 125
.sectionSize
operator, 125
.sectionStart
operator, 125
semi-hosted, 89
sequence point, 54
signed char
option, 44
simulated I/O, 89
size
of double, 16
of float, 16
size of double
option, 43
size_t type, 52
small
data model, 16
.space
directive, 122
SREC
output, 151
stack
section, 144
size considerations, 26
specify size, 144
usage, 26
startup, 96
custom, 96
statement expression, 63
static variables, 20
streams, 93
structure type, 52
stubs interface, 89
substitutions
inline assembler, 111
supported devices, 13
switch
section, 23, 82, 84
switch control
option, 45
symbol
_NearBaseAddress, 146
symbols
assembler, 114

global, 120
quoted, 114
required, 121
syntax, 114
weak, 120
syntax
assembler, 113
system include search
option, 42, 134
system startup, 96
customizing, 96

T

target
Foenix, 47, 135
option, 47, 135
target selection
option, 47, 135, 157
task
attribute, 61
termination
simplification, 95
text section, 81, 114
time
functions; date; functions, 97
tiny
base address, 145
type
enumeration, 53
ptrdiff_t, 52
size_t, 52
structure, 52
union, 52
type definitions, 55
type qualifier, 21
type qualifiers, 53
typedef, 55
typeof, 64
types
bool, 49
character, 50
integer, 49
wide character, 50

U

Ubuntu
installation, 3
uninstalling, 4
uninstalling
Arch Linux, 5
Debian, 4
Fedora, 5
Linux, 4, 5
macOS, 3
Manjaro, 5
Ubuntu, 4
Windows, 6
union type, 52
unsigned char
option, 44
used memory, 147

V

variables
auto, 19
dynamic, 20
global, 20
local, 19
static, 20
vector sections
option, 45
version
macros, 76
option, 42, 134
Visual Studio Code, 174
volatile
access size, 54
inline assembler, 109
volatile objects, 53

W

warning
pragma directive, 68
warnings
compiler, 38
controlling, 38
weak symbols, 120
option, 45, 135
wide character type, 50
Windows
installation, 5
uninstalling, 6
.word
directive, 122
.word1
relocation operator, 125
.word2
relocation operator, 125

Z

zfar
section, 23, 82, 84
znear
section, 23, 82, 84