Calypsi C compiler guide for the 68000

Release 3.5.1

Håkan Thörngren

Jan 04, 2022

Contents

CHAPTER 1

Introduction

Welcome to the Calypsi C compiler tool chain for the 68000. This guide provides reference information that can help you use the compiler tool chain suite in the best way for your application needs. There are also suggestions on coding techniques and information about the design choices and implemenation considerations made in producing 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

runtime is covered by the license of the Calypsi C compiler tool chain. They do not impose any royalties or obligations as long as they are used together with this product.

The Calypsi C compiler tool chain uses several open source components internally. These licenses can be found in the `"Licenses/Internal component licenses/"` directory in the installation directory. These do not carry any obligations over to any application built with the Calypsi C compiler tool chain.

The 64-bit floating point library routines are implemented using the Berkeley SoftFloat library which is covered by a license that can be found in `Licenses/Berkeley SoftFloat License` in the installation directory. This license only applies when you are using the `long double` type or the `--64bit-doubles` command line option. This license carry the obligation to include its copyright and license with a product makes use of the Berkeley SoftFloat library.

Note: The run-time 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

On macOS the provided installer installs the tools under `/usr/local`. Simply double click on the installation package to install the software.

After the installation you will find the installed files in a directory named `/usr/local/calypsi-68000-3.5.1`, where `3.5.1` is the version number installed. As the version number changes with every release of the Calypsi C compiler tool chain, a previous installation will not get overwritten.

In order to make it easy to invoke the tools, the installer script will add symbolic links from `/usr/local/bin` to the installation directory. This means that you only need to ensure that `/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 will also create a symbolic link from `/usr/local/lib/calypsi-68000` to the installation directory which makes it easy to refer to the installation without having to specify a version number.

2.1.1 Setting the PATH

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

2.1.2 Uninstalling

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

```
$ sudo /usr/local/lib/calypsi-68000-3.5.1/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. It may also be possible to install on an other Debian based systems, but this has not been tested.

The package can be installed from the command line using:

```
$ sudo gdebi calypsi-nut-3.5.1.deb
```

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

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

This will install the Calypsi C compiler tool chain in /usr/local/lib/calypsi-68000-3.5.1, where 3.5.1 is the version number installed.

In order to make it easy to invoke the tools, the installer script will add symbolic links from /usr/local/bin to the installation directory. This means that you only need to ensure that /usr/local/bin is in your PATH environment variable to use the most recent installed version of the Calypsi tool chain. The installer will also create a symbolic link from /usr/local/lib/calypsi-68000 to the installation directory which makes it easy to refer to the installation without having to specify a version number.

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

```
$ sudo dpkg -i calypsi-nut-3.5.1.deb
```

This may fail if there are package dependencies that need to be installed first. In this case the output will give some advice on what you need to do before you can retry the installation.

2.2.1 Uninstalling

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

```
$ sudo dpkg -r calypsi-nut
```

2.3 Arch and Manjaro Linux

Manjaro is based on Arch Linux and uses pacman. The package file can be installed from the command line using:

```
$ sudo pacman -U calypsi-68000-3.5.1-x86_64.pkg.tar.zst
```

This will install the Calypsi C compiler tool chain in /usr/local/lib/calypsi-68000-3.5.1, where 3.5.1 is the version number installed.

In order to make it easy to invoke the tools, the installer script will add symbolic links from /usr/local/bin to the installation directory. This means that you only need to ensure that /usr/local/bin is in your PATH environment variable to use the most recent installed version of the Calypsi tool chain. The installer will also create a symbolic link from /usr/local/lib/calypsi-68000 to the installation directory which makes it easy to refer to the installation without having to specify a version number.

2.3.1 Uninstalling

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

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

2.4 Windows

On Windows the provided installer installs the tools in the default folder for program files, typically `C:\Program Files (x86)`. Simply double click on the `.msi` installer to install the software.

After the installation you will find the installed files in the `C:\Program Files (x86)/calypsi-68000` folder.

2.4.1 Setting the PATH

In order to use the tools you will typically need to add `C:\Program Files (x86)/calypsi-68000\bin` to the search path. One way of doing this is to type path as a search text in the start menu, and select the suggestion Edit the system environment variables - control panel.

2.4.2 Uninstalling

To uninstall the Calypsi C compiler tool chain, simply bring up Add remove programs and select the software you want to install from the list of installed programs.

2.5 Using multiple versions

When upgrading to a newer version of the software product on a Linux operating system, the package manager will remove any previous version of the installed software. This does not apply to macOS where previously installed versions are left behind.

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.

You can use such copied installation by locally adjusting the `PATH` environment variable, or by using an explicit path in the command line.

To remove a copy of the product that has been copied elsewhere, simply delete it. Do not run the `uninstall` script as it will affect the copy installed the normal way.

Note: The license of the software product is used under a license which states the terms under which you can copy the product.

CHAPTER 3

Getting started

This chapter gives an overview how to use the compiler and the related tools.

3.1 C language

The C language is one of the most commonly used programming language in the world. While C is ideally suited for programming close to the hardware, it is also useful as a generic high-level programming language. C provides a useful set of simple but fairly powerful abstractions over a generic target machine which provide you with good control and make it possible to produce efficient applications.

The C language variant implemented here is the ISO/IEC 9899:1999 standard, which is often called the C99 standard. In this guide it is referred to as *Standard C*.

3.1.1 Cross compiler

The compiler is a cross compiler which means that it runs on a modern workstation computer, but produces applications intended to execute on a more constrained target machine.

3.1.2 Supported devices

The Calypsi C compiler tool chain supports the Motorola 68000.

3.2 File extensions

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

Table 3.1: 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`	Module page as raw data (16-bit words)
`.raw`	Raw output
`.pgz`	Foenix binary format

3.3 Building applications

You can build your applications from source files and libraries.

Source files are either written in C or assembly language which are compiled into object files. C source files are compiled with `cc68k` and assembly source files are compiled with `as68k`.

A library is a collection of object files that can be produced using the `nlib` librarian tool. This essentially combines object files together with an index, to a single file. The provided C runtime library is an example of such library. In addition you can use other libraries from third parties.

The `ln68k` linker takes object files, libraries and placement rules as input. Based on the inputs the linker puts together the executable application.

3.3.1 Compiler

The command line interface of the compiler takes a single source file and produces an object file:

```
$ cc68k source.c
```

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

Note: You will normally need use various command line options to the compiler, to select actual CPU or runtime models, among other things.

3.3.2 Assembler

For C projects you are not required to know assembly language. The whole idea with C is to allow you to write programs easier and be able to move between different architectures.

In certain situations you may want to provide very specific target code or need control at a very deep level, or to code some very critical routine using specific code. The assembler makes it possible to do this.

The command line interface of the assembler looks very much the same as for the compiler. The main difference is that the file extension differs:

```
$ as68k source.s
```

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

Note: You may need to provide a `--core` option with the assembler to make it recognize the exact set of machine instructions of the target you are using.

3.3.3 Linker

The linker is used as a separate final step to bring all object files and libraries together into an executable application.

In order to do this the linker needs a rules file needed that describes the memory system and provides placement rules for code and data. In this file you can also specify the size of the stack and the heap.

From the command line you can run the linker as:

```
$ ln68k myfile1.o myfile2.o clib-68000-lc-sd.a rules.scm
```

This is a rather simplified, but fully possible command line. The linker will produce and output `aout.elf` if executed this way that contains the application as 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

!additionalLinkerOutputs

- With the `--debug` option you can include DWARF debugging information in the ELF executable.

The linker can also produce a list file, optionally with cross reference information. This makes it possible to see how much memory is used, where things are placed in memory and also what has been included from libraries and why it was included.

3.4 Configuring

You will most likely want to consider and tune code generation of the compiler using a number of different parameters. These are specified using command line options.

The most basic settings to consider are:

- The actual core (`--core`) being used. This controls the exact instruction set.
- The data model which affects how data are placed in memory and accessed
- Size of the `double` floating point type
- Optimization settings
- Certain specific runtime environment settings, e.g. ability of the `printf` formatter

3.4.1 Core

The Calypsi C compiler tool chain currently supports the base 68000 variant.

3.4.2 Code model

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

3.4.3 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.

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 3.2: 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

3.4.4 Size of double

The `double` floating point type uses the IEEE 754 format. It can be set to either 32 or 64 bits using `--32bit-doubles` or `--64bit-doubles`. If not specified it defaults to 32 bit doubles. The `float` type is always 32 bits.

3.4.5 Optimization

You can select the level of optimization using the `-O` flag which takes a numeric argument `1` or `2`.

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

The optimization option controls additional optimization that will further reduce the memory footprint of the code and typically make it run faster.

3.5 Low level control

This section briefly covers how you can control access to specific memory and built-in functions, also called intrinsics.

3.5.1 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.

3.5.2 Intrinsics

Intrinsics functions look like ordinary function calls, but are actually special constructs to the compiler that makes it emit very specific instruction sequences.

To enable the use of intrinsics you need to include the header file `intrinsics68000.h`.

3.5.3 Assembly code

It is possible to write functions in assembly language by following the C calling convention. Such functions can be called from C in the same way as any C function.

CHAPTER 4

Data storage

This chapter discuss how you can control where data is placed and how that may affect efficiency of your application.

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 are 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.

4.1 Ways to store data

Data can be allocated as *auto*, *static* and on the *heap*. The scope of variables and need to dynamically allocate data at runtime affects how you want to allocate the data.

The general rule is that should use auto allocated variables whenever possible. This gives the compiler the most flexibility for allocating resources closest to the core, which tends to be the most efficient way to access the data.

4.1.1 Auto variables

Auto variables are function parameters and local variables in functions (that are not defined with the `static` keyword). The compiler will try to allocate such variables in processor registers!pseudoRegisters. If this is not possible, the stack is used.

Auto variables are only allocated where they are actually used. The register used for an auto variable can immediately be reused for other auto variables or temporary data as soon as the variable is no long needed. This also applies to the stack area where locations also can be reused.

Auto variables can have multiple *live ranges*, that is, a variable with the same name can be used multiple times in distinct separate parts of a function. Such variables are internally treated as different and may be allocated to different locations. Between such live ranges (or uses) they may not even exist.

When a function exits, all auto variables associated with the function are gone.

Note: If you take the address of an auto variable it means you now have a pointer to it that you can pass to other functions. This can be very useful for allocating some temporary storage and have it filled in by a function you call. However, you have to be careful to not use such pointer as the auto is deallocated when the function it is defined in exits. Also be aware that when you take a pointer to an auto, it will be allocated on the stack and it may be more costly to access it compared to another auto that has not had its address taken.

4.1.2 Static variables

A global, module or function static variable is allocated in the global memory and will take up space for the duration of the program.

The variants differ only in visibility. A global variable is visible everywhere, but to actually use it you need to make it visible using the `extern` keyword. A module static variables is visible in a compilation unit and is declared at file scope level with the `static` keyword. It is also possible to have a static local variable in a scope inside a function. Such variable is only visible inside that function when it is in scope and it will retain its value when you call that function the next time. Use the `static` keyword to distinguish such variable from an auto variable.

4.1.3 Dynamically allocated

A dynamically allocated variable is allocated from a *heap* using the *malloc* function. This is useful when you do not know how much data you will need when the program is first started.

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.

4.2 Address spaces

The compiler provides multiple *address spaces*. An address space is an addressable area of memory with certain properties. Such properties include:

- Width of address used to point to it
- Width of the associated integral index type
- The compiler will also use different instruction sequences to access different address spaces
- An address space normally has an extension keyword, or type attribute name associated with it
- Data stored in it uses section names tied to the address space, to provide control how to place it at the linking stage

Address space attributes are always active in the compiler.

The Calypsi C compiler tool chain 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.

4.2.1 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.

4.2.2 Far address space

The *far* address space makes use of the full 32 bits 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.

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.

4.2.3 Summary

The following table summarizes the available address spaces.

Table 4.1: 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.

4.2.4 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 acts on the type it is closest to, which is `int` in this example. There is not particular order required between type qualifier and type in C, it means the same thing.

4.2.5 Pointers

A pointers 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;
```

In this case p1 is a pointer stored in default memory that points in an int which is stored in Far memory memory.

p2 is a pointer stored in Far memory memory that points to a long in default memory.

4.2.6 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 points 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 5

Placing code and data

When compiling a program on a modern hosted machine you do not really need to care much about placement of code and data. The tools will handle it for you and this works well as you are running on the same machine as you are targeting.

It starts to become more complicated for a cross compiler as it needs to know which target system you are compiling for. It gets even more complicated if you target an embedded system, as the memory system varies wildly and you often need good control over how code and data are placed in different memories.

Good tools for embedded system gives you control over how to place the code and data. This is done by using building blocks that gives flexibility and control over placement. The compiler is able to help you by providing abstractions that makes it easier to express the memory system, mainly by the means of type qualifiers, initializers and run-time model settings.

5.1 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 takes care of placing data objects and functions in suitable sections following certain rules. For data objects the compiler looks at if the data object is declared `const`, uses a specific address space and whether it has explicit initializers.

Table 5.1: 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

The sections `inear`, `ifar`, `ihuge` 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 certain sections to ROM and others to RAM. This is how it works for a ROM based application where power is applied and the application starts. When building an application for a hosted system where it is loaded from some storage media to RAM, this distinction may not apply. In any case, you should regard sections marked as ROM to be read-only.

The section types are carried over from the UNIX world, refer to *Section kinds* in the assembler reference for more information.

Note: When generating an application that can execute from ROM, the linker will separate an initialized data sections into two sections. The initializers go into a new section which is prefixed by an `i` that should be placed in ROM. The real data section is placed in RAM. The linker will then arrange with the help of the C run-time startup module to copy the initializers from ROM to RAM before giving control to the `main()` function.

5.1.1 Linking process

The linker reads object files with sections and a placement rules file. The placement rules file describes available memory areas and what sections that can placed in them, possibly with additional placement constraints.

The memories that contain actual value bits, which can be program code or initialized data are emitted in the executable.

An executable can either be used as input to some ROM or FLASH programmer, be loaded by the target machine or into a debugger which connects to, or simulates a target machine internally. There are different executable formats and you can choose one that fits the needs of the execution environment.

Note: Even though code a data are completely separated by using different sections, it may be desirable to place such sections together in the same memory. This can be done at the link stage by placing such sections in the same memory.

5.1.2 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 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))
            (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))          ; user stack
    (block sstack (size #x200))          ; supervisor stack
    (block heap (size #x4000))           ; heap
    (base-address _NearBaseAddress NearRAM #x8000)
    ))
```

The rules file is evaluated by a Scheme interpreter. In this simple form it is just a definition of a global data object named `memories`. This contains a list of `memory` objects that have a name, an address range and a list of sections that goes into a given memory. There are ways to specify that certain sections are to be placed at a fixed address. Finally there are blocks that defines sections related to the stack and heap. These just create a section and specify the size. Such block sections also need to be tied to a memory in the sections list.

Note: As the linker rules file is actually a Scheme program that is evaluated, it can in theory construct the `memories` object by running functions or expanding macros. The expected outcome after evaluating the linker rules source file is that the Scheme interpreter leaves an object named `memories` in its global environment when it terminates. However, you will probably do not need anything more than a rather simple file as in the example shown.

5.2 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.

5.2.1 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: If you are running a hosted environment you can specify the command line option `--hosted` to the linker. In such case it is assumed that the program image is loaded into RAM from some external media. In this case initialization of variables can be done in-place by the load action. Thus, having separate initializers to be copied is not needed.

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

5.3 Stack

The stack is where function return addresses, saved registers and variables are kept.

The stack helps keeping track of the current execution state and makes it possible to implement reentrant and recursive functions.

5.3.1 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 essentially 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 6

C language

This chapter discusses the C language as it is implemented. The language supported is the ISO/IEC 9899:1999 standard, which is often called the C99 standard.

6.1 Overview

If you are used to older versions of the C, the C99 introduces several new 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

6.2 Extensions

Language extensions are always enabled and include various qualifier keywords, intrinsic functions and some additional features available in the library.

CHAPTER 7

Efficient coding

The Calypsi C compiler tool chain is designed to be used when coding for memory constrained systems, typically embedded systems. This discusses how you should think about using proper data types, how to think on both smaller as well as larger targets.

Even though your coding today is for a certain system, the code may live on and be moved to either larger or smaller systems. This chapter discusses tradeoffs between such environments.

7.1 Data types

Selecting data types have great impact on the overall program size and execution speed. The following are rules of thumb:

- In general you should use as small types as possible
- Floating point operations are in general inefficient, they occupy space and are often slow
- Use variables with as narrow scope as possible
- Avoid taking the address of a variable. If you do, limit its use as much as possible. The reason is that the compiler cannot allocate such variable in a register (there is no address of a register). It also means the optimizer cannot know how the variable is affected by function calls

7.1.1 Integer types

Even though using the smallest possible type is recommended, it has the drawback that if the code is compiled for a larger target, it may actually be more efficient to use `int` and `unsigned int`, as it is often the sweet-spot in terms of efficiency.

The `stdint.h` header file contains a number of integer types that are very helpful in select appropriate integer type based on size, efficiency and portability.

A type such as `uint_fast16_t` defines an unsigned type of at least width 16 bits that is meant for fast execution. An 8 bit target will set this to a 16 bit type, while a 32 bit target may choose to use a 32 bit type for it, especially if the instruction set is better at handling 32 bit operations compared to 16 bit operations.

7.1.2 Floating point types

Floating point is often implemented using library routines in assembly language which occupy code space and have execution overhead.

One alternative is to use normal integers in fixed point instead. These are often described in Q notation. For example, a Q24.8 number has 24 bits of integer part and 8 fractional bits. Such numbers can be manipulated using normal integer operations and still represent fractional parts. You will need to compensate and scale values when operating on them, which is reasonable straightforward.

7.1.3 Implicit conversions

C has conversion rules which strives to convert integers to `int` and `float` to `double`. This can introduce additional code and if this is a concern, you either need to understand the rules for such conversions, or study the generated code from the compiler.

Note: While studying generated assembly code may sound daunting, you do not actually need to understand what is going on in detail. Having some familiarity and a rough feeling for how it works will take you a long way. The compiler can be requested to generate a list file (`-l` or `--list-file` command line option). The resulting list file will show the C code intermixed with the generated assembly code.

7.2 Use prototypes

Functions can be declared in two ways. Traditional C uses a style without prototypes that is not recommended as it makes it harder for the compiler to detect errors in the code. Using prototypes is safer and allows for better code generation as it avoids certain type promotion rules that can introduce extra code. The traditional style is only supported for compatibility reasons with old code. In traditional style a declaration does not specify the 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, but that is promoted to `int` during function call and causes additional code.

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

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

```

(continues on next page)

(continued from previous page)

```c
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.

7.3 Use tables

Consider the tradeoff between calculating something compared to using a table to encode some knowledge or operation.

If implementing a sine operation in Q (fixed point) notation, that can be approximated using a table lookup rather than calculated, which can make the operation very fast and compact.

CHAPTER 8

Running the compiler

The interface to run the compiler is command line based. It is also possible to use the compiler from an IDE. In this case the IDE interfaces to the compiler using the command line.

8.1 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 an argument. The argument appears after option, they can be separated either by a space or an equal (=) sign. For single argument options, the argument is allowed to appear without any separator:

```
$ 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 `-v` or `--version`:

```
$ cc68k --version
Calypsi ISO C compiler for Motorola 68000 version 3.5.1
```

Command line options are described in *Command line options*.

8.2 Include search path

Header file inclusion exists in two forms. You can either surround the filename with double quotes or angled brackets:

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

The angled form is normally used for system header files while the quoted form is used 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

A header file specified in angle brackets are not searched relative to the compilation directory, otherwise they behave identical.

The system header file directory is the installation directory.

Note: More precisely the system header file directory is a directory relative to actual cc68k executable file. This means that if you move an installation to another location in a the file system, it will still be able to find the correct system header file directory.

8.3 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.

8.3.1 Object file

The object file is in the ELF format with optional DWARF debugging information (if `--debug` (also named `-g`) option is specified).

The object file contains the following components:

- A symbol table
- Relocatable sections with code and data
- Relocations, to allow address values to be altered by the linker when the actual value is know after placing the section
- Source level debugging information in DWARF format (if the `--debug` command line option was specified)
- Vendor specific data, which carries certain additional information specific to the Calypsi C compiler tool chain. This includes runtime attributes as well as additional information not carried by ELF/DWARF, but that is used by the linker and debugger.

8.3.2 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 array summation function:

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

If compiled with the `-l` option it will produce a list file that looks as follows:

```
################################################################################
#                                                                              #
# Calypsi ISO C compiler for Motorola 68000                      version 3.5.1 #
#                                                        04/Jan/2022  21:35:59 #
# Command line: sum.c -l                                                       #
#                                                                              #
################################################################################


    \ 00000000                      .rtmodel version,"1"
    \ 00000000                      .rtmodel nearDataBase,"A4"
    \ 00000000                      .rtmodel core,"68000"
    \ 00000000                      .rtmodel codeModel,"large"
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
```

8.4 Diagnostics

Compiler diagnostic messages are given for a number of different reasons. They identify problems that either must, or should be addressed in some way.

8.4.1 Errors

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

8.4.2 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.

8.4.3 Internal errors

An internal error may result if the compiler detects a problem internally that should not occur. This always indicates that there is an error in the compiler product.

8.4.4 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 the compare uses an assignment, this is a common error in C. Maybe you intended to use equal operator (==), however it is perfectly legal to test the result of an assignment expression is non-zero. The compiler cannot know your intention, but there is a potential problem here. Compiling this code will result in some warnings:

```
w.c:3:10: warning: using the result of an assignment as a condition without parentheses [-Wparentheses]
   if (x = 6) {
       ~~^~~
```

(continues on next page)

(continued from previous page)

```
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 identifies 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.

You can add an extra pair of parentheses around the assignment, which is generally considered a good way of stating that this is an intentional assignment inside 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 equal operator, you should change it to use the correct operator:

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

8.5 Command line options

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

8.5.1 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 [-v|--version] [-o|--output-file OUTPUT-FILE] [-l]
             [--list-file LIST-FILE] [-I DIRECTORY] [-D IDENTIFIER]
```

(continues on next page)

(continued from previous page)

```
            [-U IDENTIFIER] [-g|--debug] ([--32bit-doubles] |
            [--64bit-doubles]) ([--char-is-signed] | [--char-is-unsigned]) [-E]
            [--print-macro-definitions] [-O LEVEL] ([--space] | [--speed])
            [--no-cross-call] ([--no-inline] | [--always-inline])
            [--only-marked-as-inline] [--strong-inline] [--rtattr NAME=VALUE]
            [--weak-symbols] [--no-vector-sections] [-c]
            [--force-switch STRATEGY] [-W ARG] [--pedantic-errors]
            [--core CORE] [--target TARGET] [--fd DIRECTORY]
            [--code-model NAME] [--data-model NAME] FILE
  use 'cc68k --help' for detailed help
```

You can request a longer help using the `--help` option:

```
$ cc68k --help
Calypsi ISO C compiler for Motorola 68000 version 3.5.1

Usage: cc68k [-v|--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] [-O LEVEL] ([--space] | [--speed])
            [--no-cross-call] ([--no-inline] | [--always-inline])
            [--only-marked-as-inline] [--strong-inline] [--rtattr NAME=VALUE]
            [--weak-symbols] [--no-vector-sections] [-c]
            [--force-switch STRATEGY] [-W ARG] [--pedantic-errors]
            [--core CORE] [--target TARGET] [--fd DIRECTORY]
            [--code-model NAME] [--data-model NAME] FILE
  use 'cc68k --help' for detailed help

Available options:
  -v,--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
  -I DIRECTORY             Include directory
  -D IDENTIFIER            Predefine a macro
  -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
  -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-inline              Disable all function inlining
```

(continues on next page)

(continued from previous page)

```
  --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'
  --rtattr NAME=VALUE      Define a run-time 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
  --force-switch STRATEGY  Switch strategy, one of 'if-else', 'jump-table' or
                           'value-table'
  -W ARG                   Warning control
  --pedantic-errors        Error on language extensions
  --core CORE              CORE, one of '68000' or '68010' (defaults to '68000')
  --target TARGET          Target system, one of 'Amiga' or 'Foenix' (defaults
                           to embedded/ROM use, if omitted)
  --fd DIRECTORY           Add directory to search path for Amiga library .fd
                           files
  --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
```

8.5.2 Options in detail

`--version`, `-v`

Displays the name and version of the compiler.

`--output-file`, `-o`

Use this option to specify the name of the output object file. If not given, the output file is created from the name of the input file (ignoring any directory path) and sets the file extension to `.o`. Thus, output files are written to the compilation directory by default.

You can use this option to alter the name of the output file as well as providing a directory path to it. Such 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`.

The `-l` and `--list-file` options are mutually exclusive, only one of them can be stated on the command line.

`--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.

The `-l` and `--list-file` options are mutually exclusive, only one of them can be stated on the command line.

-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.

-D

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

-U

Undefine a preprocessor symbol. 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.

-g

Synonym for `--debug`.

--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

This makes the data type `char` signed. The default is that `char` is unsigned.

--char-is-unsigned

This makes the data type `char` 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

Run the optimizer. This expects and argument that is either 0, 1 or 2. `-O0` (the default) means no additional optimization passes. This still performs a decent amount of work to produce good code. `-O1` enables additional optimizer passes to further improve the code. Finally `-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 control 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

Disable the cross call optimizer which is normally enabled at `-O2`. The cross call optimizer lifts out common code sequences to small subroutines. This can have a very beneficial impact on code space, so it is enabled by default when using `--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 consider functions marked with the `inline` keyword for inlining. This will disregard small functions and single use static functuions from being considered for inlining.

--strong-inline

Regard functions marked with the `inline` keyword as a strong hint that 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 or to select a particular variant of a function that exists in many variants.

The `printf()` formatter is an example of this. Three variants with identical names are included in the C library, they are separated by having different runtime attribute values which makes it possible to select which formatter variant to use during the link stage.

`--weak-symbols`

Make all public symbols in the object file weak. This is normally not needed, but can be used in certain situation to make a default implementation of some function in a library. This makes it possible to override the function with one that is not weak, or fall back to use the default one when 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 and do not try to link. This is actually a no-operation as the compiler always does that. This option is provided as this is such a commonly used option with many compilers.

`--force-switch`

This tells the compiler to use a specific strategy for generating switch tables. The compiler normally uses heuristics to determine how to generate a switch table. In some cases you may want to override the heuristics and force a particular variant.

The variants provided are series of if-else tests, a jump table or a table with value and label pairs. For very small switches the if-else approach is often best. The jump table is typically the fastest when having a decent amount of entries that are close (or next) to each other. Finally, the value and label pairs work well for tables of some size where the values are spread out. This uses a larger table and a function that performs a binary search lookup which means it scales fairly well.

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

`-W`

Control warnings, see *Controlling warnings*.

`--pedantic-errors`

Generate errors on using language extensions. The language supported has certain relaxations to the strict C standard intended to make the language a little bit more flexible. Use this option to disable such extensions.

`--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 only known specific platform is Foenix. This tells the compiler to generate code for the Foenix hardware math unit and enables hosted behavior.

`--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 9

Data representation

This chapter describes the available data types in detail.

9.1 Basic types

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

Table 9.1: 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 $2^{64}-1$

9.1.1 bool

To use the bool data you need to include `stdbool.h` which also defines `true` and `false`.

The boolean data type is also available as `_Bool` without having to include `stdbool.h`.

9.1.2 char

The character type is unsigned by default. You can enable signed `char` by compiling the code with the `--char-is-signed` command line option. Note that the supplied C library has been compiled with `char` as unsigned.

Note: The `char` type differs from `short`, `int` and `long` in that it defaults to being unsigned here. 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 int `stdint.h`.

9.1.3 wchar_t

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

9.1.4 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.

9.1.5 Floating-point types

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

Table 9.2: 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 run-time 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.

Note: When the Foenix tareget is enabled (using `--target=Foenix` command line option), the built-in hardware math module is used for 32-bit floating point operations. This math module does not support subnormal numbers and all operations that would produce a subnormal number instead generates zero.

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.

9.1.6 Function pointer types

For the 68000 architecture function pointers are always 32 bits.

9.1.7 Data pointer types

The following data pointers are available:

For the 68000 architecture data pointers are always 32 bits.

Each data pointer has an associated index type which is always a signed integer type. This type is used in address calculations, e.g. array access with an index, or when advancing a pointer by adding (or subtracting) an integer value.

9.1.8 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.

9.1.9 size_t

This is an unsigned integer type that holds the largest size of an object.

On the 68000 the size of `size_t` is 32 bits.

9.1.10 ssize_t

This type has the same bit width as `size_t`, but has a signed type. This type is used by library routines.

9.1.11 ptrdiff_t

This is a signed integer type that represents a distance inside the largest possible object.

On the 68000 the size of `ptrdiff_t` is 32 bits.

The value that results from sutracting two data pointers (pointing inside the same object) has type `ptrdiff_t`. The value itself represents 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 can result in a negative result because the value is larger than what can be represented by `ptrdiff_t`.

9.2 Structure types

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

9.3 Union types

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

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

int main() {
  struct Scope x;
```

(continues on next page)

(continued from previous page)

```
  x.u.num = 65;
  x.u.alpha = 'A';
  return 0;
}
```

The use of a declarator on the union inside the structure means that it takes an extra step to access its members. This has been relaxed in the C11 standard so that you can instead 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 is enabled by default and allows such anonymous unions to be compiled without any diagnostic messages, even though the compiler actually supports C99.

You can enable warnings if you use such extensions by adding the command line option `-Wc11-extensions` or `-Wpedantic` to the command line.

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

9.4 Enumeration types

Enumeration types are represented as `int`. If you want to use a smaller storage representation you need to pick a smaller integral type. If you want a better name for it, you can use a typedef:

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

typedef char fruit_t;

fruit_t active;

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

9.5 Type qualifiers

Standard C provides two type qualifers, `volatile` and `const`.

9.5.1 Volatile objects

C has a concept of volatile objects. They are typically used for accessing hardware. Both writing and reading volatile objects are seen as side-effects that will happen.

Somewhat related there is a concept of *sequence points* in a program. A volatile accesses that is performed between two sequence points will take place between those sequence points. It is not allowed to move any volatile access past a sequence point.

To ensure that two memory accesses occur in a certain order, you need to make them volatile and separate them with a sequence point. The semi-colon 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 that is wider than the natural register size of the target will result in that the access is performed in several steps as dictated by the register size of the target machine.

Reading and only using a portion of a scalar volatile object will result in that the whole object is 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 just read the lower 16 bits of `wide`.

Assignment results

C has the notation that an assignment has an expression value, consider:

```
volatile int var;

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

The assignment will write 4 to the volatile, but what will the function actually return? It could either (always) return 4 or it could whatever `var` reads to after the assignment. The answer is that it is implementation defined and could be either.

You should in general avoid such constructs in your programs and be more explicit with what you want to happen. 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 makes the intention of the code clear and has the benefit that it will the same behavior no matter which C compiler you use.

Bit fields

There are no guarantees what will happen when accessing volatile bit fields. A bit field typically describes a subset of bit locations inside its storage unit. In many cases adjacent bits may get accessed, depending on layout and memory storage units. Using bit fields together with volatile 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.

9.5.2 Const objects

The `const` type qualifer indicates that an object is read-only. It can be applied to data objects as well as pointers.

When defining a static object as `const`, the compiler will try to place it in a section that is read-only.

A pointer to a `const` object can point to both a read-only as well as writable objects. Using such pointer indicates that any user of the pointer should not attempt to alter memory. This can help the optimizer. It is also considered good practise as it prevents altering data when the some part of the applicaton is not intended to do writes to the data object.

9.6 Type definitions

Using type definitions (the `typedef` keyword) in your code is a highly recommended practise that has several benefits:

1. It makes your code more readable when types have meaningful names. A `speed_t``gives a lot more understanding than ``long`. Also if you need to change the definition of it, there is only one place where it needs to be done. This saves you from going through the code to find all the `long` and change them to something else. Not to mention that you should only change those `long` in your code that actually are speed and not any other uses of `long`.
2. A `struct fish` can be named `fish_t`, which is shorter and just as readable. It also hides that it is a structure, which may be desirable in many cases

There is no cost related to using `typedef`, the generated code will be the same.

9.7 Alignment

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

CHAPTER 10

Extended attributes

An *attribute* is some additional that can be attached to a function, data object or type. You can attach an attribute by specifying it as a keyword. Standard C has some built-in keywords, e.g. as `const`.

Extended attributes are used to gain gain access to certain behaviors or properties that are not part of Standard C. They are either tied to the specific target system or are features that are useful for embedded systems.

10.1 Overview

Attribute can be applied either with keyword syntax or using attribute syntax. A keyword is just a single word, e.g. `__far` while the attribute form looks as `__attribute__((far))`. They are functionally the same, but there are certain situations where the C parser will not accept the keyword form and you may need to use the attribute form.

Otherwise it is mostly a matter of taste which you use. You may also want to use preprocessor defined macros to rename attributes to make the code more portable. Such preprocessor macro definitions can also be used to turn some attributes off when compiling for another target, or rename it to match another compiler or different target.

10.2 Using attributes

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

10.2.1 Syntax for data objects

You can apply an attribute to data objects in the following way:

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

The location of the attribute does not matter when applied to an object. The example above will apply the Far memory attribute to all objects defined (a, b, c and d).

10.2.2 Syntax for pointer types

Attributes can also be applied to pointer types. In this case the location of the attribute is significant as it affects what it applies to. It can either say that a pointer is constant, or that what it points to is constant.

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

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

In this case p1 is a pointer stored in default memory that points in an int which is stored in Far memory memory.

p2 is a pointer stored in Far memory memory that points to a long in default memory.

10.3 Attribute reference

This section goes through all available extension keywords and attributes.

10.3.1 Summary of attributes

The following table summarizes available attributes. To use an attribute in keyword form you need to add two underscores to it, e.g. __far rather than just far. When used in attribute syntax the attribute name is used together with __attribute__, e.g. __attribute__((zpage)).

Table 10.1: Extended attributes summary

Attribute name	Description
aligned(nn)	Specify alignment of data object or function
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
saveds	Entry point that needs to initialize base pointer in register A4
intrinsic	Used to declare an intrinsic function
task	Relaxes preserving registers

10.3.2 Description of attributes

This section describes each attribute in detail.

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 is specified is given 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 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 `intrinsics68000.h` file which contains all such valid declarations.

`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 11

Pragma directives

The pragma directives is a mechanism in standard C which allows vendor specific extensions to the C language, while still keeping it portable to other C compilers.

You can use pragma directives to control and direct the behavior of the compiler. This can include suppressing certain warnings or to control placement of an object to a certain memory area by using a custom section.

The pragma directives are always enabled by the compiler and you can use them either by the `#pragma` preprocessor directive or the `_Pragma` preprocessor operator.

11.1 Pragma directives reference

This section goes through all available pragma directives.

11.1.1 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

Note: As the C front end used is Clang, the same names are used when a pragma directive works the same way. This is done to make it easier to switch between compilers, even if pragmas are vendor specific extensions.

11.1.2 Description of pragma directives

section

The `#pragma clang section` directive makes it possible to assign section names to functions, global and static objects.

The compiler will normally place global and static objects using predefined section names. In some cases you may want to override it 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 previously one seen is used.

The `section` attribute takes precedence over the `clang section` pragma directive. This means that if a section name is specified using `__attribute__((section(myname)))`, it has precedence.

Note: No interpretation of the section name is done. 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 what diagnostic messages are enabled through the use of pragmas in the source code. This is useful for turning off specific warnings in a section of source code.

The pragma may control any warning that can be used from the command line. Warnings may be set to ignored, warning, error, or fatal.

In addition you can also push and pop the current warning state. This is particularly useful when writing a header file that will be compiled by other people, because you dont know what warning flags they build with.

In the following example `-Wextra-tokens` is ignored for only a single line of code, after which the diagnostics return to whatever state had previously existed.

```
#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 can be used to ensure that some other module is included at the link stage even if you do not call or reference any function or data object in it.

This is mainly intended to be used in libraries that can request code slices or objects to be included. One example when this is used is in the heap system provided by the C library. If you use a function such as `malloc()` there is a `require` pragma directive present in that module which will enable code in the C startup to initialize the heap:

```
#pragma require __call_heap_initialize
```

CHAPTER 12

Intrinsic functions

This chapter covers the available predefined intrinsic functions and built-in functions.

Intrinsics functions look like ordinary function calls, but are actually special constructs to the compiler that makes it emit very specific instruction sequences.

To enable the use of intrinsics you need to include the header file `intrinsics68000.h`.

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

12.1 Intrinsic functions reference

The `intrinsics68000.h` file looks as follows:

```
#ifndef __INTRINSICS_68000_H
#define __INTRINSICS_68000_H

#ifdef __CALYPSI_TARGET_M68K__

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);

__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();
__attribute__((intrinsic)) void __break_instruction(unsigned char);
```

(continues on next page)

(continued from previous page)

```
#endif // __CALYPSI_TARGET_M68K__

#endif // __INTRINSICS_68000_H
```

12.1.1 Summary of intrinsic functions

Table 12.1: Intrinsic functions summary

Function name	Description
`__disable_interrupts`	Disables interrupts
`__enable_interrupts`	Enables interrupts
`__get_interrupt_state`	Gets the current interrupt status
`__restore_interrupt_state`	Restores to a previous interrupt status
`__get_return_address`	Gets the return address of the current function
`__set_return_address`	Alters the return address of current function
`__break_instruction`	Generate a `BKPT` instruction
`__reset_processor`	Generate a `RESET` instruction
`__stop_processor`	Stops 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

12.1.2 Description of intrinsic functions

This section describes each intrinsic function in detail.

`__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 status 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 status 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 status, 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 13

Preprocessor

This chapter provides a brief description of the preprocessor and covers macros that are most commonly used.

13.1 Overview

The preprocessor adheres to Standard C and provides the following:

1. Predefined macros that makes it possible to inspect the compilation environment. This allows you to fine tune your application based on properties provided by the compiler.
2. User defined macros, from the command line or set inside C source files.
3. Ability to dump the preprocessor environment in order to inspect it.
4. Ability to dump the source file as it is after running the preprocessor

13.1.1 Predefined macros

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

__STDC__

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

__STDC_HOSTED__

This macro is defined to 0 to indicate that this is a cross compiler. You can test this using:

```
#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 to indicate that either the assembler or compiler being used is by Calypsi.

`__CALYPSI_CC__`

This is set to 1 to indicate this compiler is a Calypsi C compiler.

`__CALYPSI_ASSEMBLER__`

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

`__CALYPSI_VERSION_MAJOR__`

This macro is set to the first number in the version number triplet.

`__CALYPSI_VERSION_MINOR__`

This macro is set to the second number in the version number triplet.

`__CALYPSI_VERSION_FIX_LEVEL__`

This macro is set to the third number in the version number triplet.

`__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. It has the same value as one of `__ORDER_LITTLE_ENDIAN__`, `__ORDER_BIG_ENDIAN__` or `__ORDER_PDP_ENDIAN__`.

Normally it is easier to use the `__BIG_ENDIAN__` or `__LITTLE_ENDIAN__` macros to conditionally include source code that is dependent on the endian behavior of the target.

`__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 to 4321 to describe the byte order in memory for a big endian target.

`__ORDER_PDP_ENDIAN__`

This macro is defined to 3412 to describe the byte order in memory for a PDP endian target.

`__CALYPSI_TARGET_M68K__`

This macro is set to 1 to indicate that the target is the 68000 family.

`__CALYPSI_CORE_68000__`

This macro is set to 1 to indicate that the selected core is 68000. You can test if the core is 68000 using `#ifdef __CALYPSI_CORE_68000__`.

`__CALYPSI_CORE_68010__`

This macro is set to 1 to indicate that the selected core is 68010. You can test if the core is 68010 using `#ifdef __CALYPSI_CORE_68010__`.

`__CALYPSI_TARGET_SYSTEM_EMBEDDED__`

This macro is set to 1 to indicate that the compiler is configured for an embedded system, generating code suitable to be put in a flash or ROM memory.

`__CALYPSI_TARGET_SYSTEM_FOENIX__`

This macro is set to 1 to indicate that the compiler is configured for the C256 Foenix.

`__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 14

Section reference

Code and data objects produced by the compiler end up in sections which are written to the object file output. Having some understanding of sections helps writing linker placement files and is also useful when studying list files produced by the compiler and linker.

14.1 Section overview

Code and data objects will in the end occupy space in target memory. This can be done in two ways, either by providing actual bits to write to memory, which is called *program-bits* in ELF. A section can also take space in target memory, but in the ELF object file it is only represented by a size, there are no actual values in the object file, these are called *no-bits* in ELF.

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.

There are three major groups of sections, *text*, *data* and *bss*. Text sections are read-only program-bits sections. Data are read-write program bits sections. Finally, bss corresponds to read-write no-bits sections.

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 object file by just storing the size of the section without any explicit data bits.

14.1.1 Section names

A section has a name that identifies it. There can be many sections sharing the same name. The purpose of the section name are twofold, one is to give it a descriptive name that identifies the section. The second is to control placement in memory based on rules.

14.1.2 Section types

Each section has a type which are the already mentioned *text*, *data* and *bss*. However, for embedded systems use, there are also *rodata* (read-only data) and *no-init*. Read-only data sections are essentially text sections, with the minor difference that it identifies it as being data and not executable code. No-init sections are like bss, but without the mechanism that clears the memory.

14.1.3 Data sections

Data sections in the view of the compiler are read-write program-bits sections, that is, they have initializer values that will end up in memory. When you use a hosted compiler, such section is simply loaded to memory before the program starts.

This will not work if the program is written to a flash memory and the program is expected to start when the power is turned on.

To solve this problem the linker clones data sections. The original data section has its initializers removed and is placed in RAM. The clone has a section name with an `i` prepended to it, e.g. if it was named `data`, it will now become `idata`. This section is intended to be put in a flash (read-only) memory.

In order to initialize data sections when power is turned on, a copy routine is inserted before the `main()` function is called. This routine copies the initializers from flash to RAM to initializer the data memory. Clearing bss sections are also taken care of by this routine.

What is copied and cleared is described in a table that is created by the linker. This table is implemented by a linker created section names `data_init_table`.

Note: On certain targets the Calypsi C compiler tool chain can be used to cross compile executable binaries intended to be loaded and executed from an operating system. This is supported by the `--hosted` command line option. The effect of this is to not clone data section as it would mean duplication. In such environment there is still a need for the table driven initializer to handle bss sections.

14.2 Sections used by the compiler

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

Table 14.1: 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

The sections `inear`, `ifar`, `ihuge` 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 certain sections to ROM and others to RAM. This is how it works for a ROM based application where power is applied and the application starts. When building an application for a hosted system where it is loaded from some storage media to RAM, this distinction may not apply. In any case, you should regard sections marked as ROM to be read-only.

14.2.1 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.

14.2.2 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 bits offset, which means all calls are assumed to be reachable.

The comp

znear

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

near

Holds zero initialized (bss) 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`.

CHAPTER 15

Runtime library

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

15.1 Design considerations

The Apache NuttX library in its original form is a conglomerate of library routines from the C standard library, POSIX standard as well as other things commonly found in real-time operating systems.

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

In this adapted version it is turned into a Standard C library where the POSIX and RTOS style parts are either removed or disabled.

Apart from that, the following are the key changes:

- Header files that are very closely related to the compiler e.g. `stddef.h`, `stdarg.h`, `setjmp.h`, `stdint.h` are replaced.
- Internal names are changed so they either start with double underscores, or single underscore followed by a capital letter. This is done to avoid polluting the namespace as a C library as such identifiers are set aside for the compiler vendor.
- The stub interface are using names prefixed by `_Stub` rather than `up_` to make it more clear what it is, but also for the reason mentioned above.

Note: The Apache NuttX library comes with a configuration utility intended to tailor it to a specific environment. This configuration utility is not used here and the build system is different. This makes the library ready to use without much of a configuration step at all. This is possible to do as the compiler used is known and target board specific matters are not part of it. Some of the library functions exist in different versions, mainly to assist in reducing the memory footprint by making certain capabilities optional, e.g. formatters, see *Library on a diet*.

15.2 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;
}
```

To link with the library you will need to add the library to the command line:

```
$ ln68k main.o clib-68000-lc-sd.a linker-rules.scm
```

The actual name of the library depends on the setting you have used with the compiler. There are runtime attributes in the object and library files that prevents object files compiled in different way that makes them incompatible from being linked together.

15.3 Provided library files

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

Table 15.1: Sections

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-Foenix.a`	Small	Small	32 bits	Foenix
`clib-68000-lc-sd-Foenix.a`	Large	Small	32 bits	Foenix
`clib-68000-sc-ld-Foenix.a`	Small	Large	32 bits	Foenix
`clib-68000-lc-ld-Foenix.a`	Large	Large	32 bits	Foenix
`clib-68000-sc-fod-Foenix.a`	Small	Far-only	32 bits	Foenix
`clib-68000-lc-fod-Foenix.a`	Large	Far-only	32 bits	Foenix
`clib-68000-sc-sd-double64-Foenix.a`	Small	Small	64 bits	Foenix
`clib-68000-lc-sd-double64-Foenix.a`	Large	Small	64 bits	Foenix
`clib-68000-sc-ld-double64-Foenix.a`	Small	Large	64 bits	Foenix
`clib-68000-lc-ld-double64-Foenix.a`	Large	Large	64 bits	Foenix
`clib-68000-sc-fod-double64-Foenix.a`	Small	Far-only	64 bits	Foenix
`clib-68000-lc-fod-double64-Foenix.a`	Large	Far-only	64 bits	Foenix

15.4 Stub interface

How to provide functions such as `fopen()`, `fprintf()` or even `assert()` when it is a cross compiler that cannot possibly know about the final execution environment of the application? It may not even have a file system.

The answer is that functions like `fprintf()` goes through some layers of library functions, that may deal with formatting, buffered I/O, some stream interface, but in the end it needs to actually output characters to some display or file. This is solved by a simple stub API which is defined in the `stubs.h` header file.

The Calypsi C compiler tool chain comes with a semi-hosted debug implementation of the stub API, which is supplied in the normal C library. To enable support for semi-hosting, add the command line option `--semi-hosted` to the linker command line. The actual stub modules are included in the C library, but they are ignored by default.

When linked with semi-hosted stubs the db68k source debugger can detect the stub actions and perform the actions in the debugger environment. This way the application can output characters to a console or to a file that is on your host computer. This can also handle things such as assert. While this is very useful for debugging purposes, you will in the end need to provide small stub functions that work with execution environment of you application for any stub functions used in the final application.

Note: Even if you enable some stub actions in you application, you can link with `--semi-hosted` to get semi-hosted support for the stub actions you do not implement. This can be used if you implement a file system, but you may still want to use the `assert()` in the semi-hosted mechanism. This works as the semi-hosted stubs in the C library have been compiled with `--weak-symbols`, which means any stub function you implement takes precedence, but in case you leave some stub unimplemented, the weak semi-hosted variant will be used.

The `stubs.h` file looks as follows:

```
#ifndef __INCLUDE_STUBS_H
#define __INCLUDE_STUBS_H

#include <nuttx/config.h>
#include <nuttx/compiler.h>

#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>

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

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

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

/****************************************************************************
 * Name: _Stub_putchar
 *
 * Description:
```

(continues on next page)

(continued from previous page)

```
 *   Output one character on the console
 *
 ******************************************************************************/

int _Stub_putchar(int ch);

/*******************************************************************************
 * Name: _Stub_puts
 *
 * Description:
 *   Output a string on the console
 *
 ******************************************************************************/

int _Stub_puts(const char *str);

/*******************************************************************************
 * Name: _Stub_raise
 *
 * Description:
 *   Handle a raised signal
 *
 ******************************************************************************/

int _Stub_raise(int signo);

/*******************************************************************************
 * 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
 *   are permissions, e.g 0666. If the file system does not handle
 *   permissions you can ignore that this function is variadic.
 *
 ******************************************************************************/

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

/*******************************************************************************
 * Name: _Stub_close
 *
 * Description:
 *   Close a file
 *
 ******************************************************************************/

int _Stub_close(int fd);

/*******************************************************************************
 * Name: _Stub_lseek
 *
 * Description:
 *   Change position in a file
```

(continues on next page)

(continued from previous page)

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

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

/****************************************************************************
 * Name: _Stub_read
 *
 * Description:
 *   Read from a file
 *
 ****************************************************************************/

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

/****************************************************************************
 * Name: _Stub_write
 *
 * Description:
 *   Write to a file
 *
 ****************************************************************************/

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

/****************************************************************************
 * Name: _Stub_ioctl
 *
 * Description:
 *   Control a device
 *
 ****************************************************************************/

int _Stub_ioctl(int fd, unsigned long request, ...);

/****************************************************************************
 * 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_assert
 *
 * Description:
 *   Handle an assertion
 *
 ****************************************************************************/

#ifdef _CONFIG_HAVE_FILENAME
void _Stub_assert(const char *filename, int linenum) __noreturn_function;
```

(continues on next page)

(continued from previous page)

```
#else
void _Stub_assert() __noreturn_function;
#endif

#if defined(__cplusplus)
}
#endif


#endif /* __INCLUDE_STUBS_H */
```

15.4.1 Custom I/O

You can provide your own low level I/O stubs. File descriptors at this levels are just integers that acts as index to some internal look up table, typically an array. The library is by default configured with six streams. The first three (file descriptor index 0-2) are the well known `stdin`, ``stdout` and `stderr`. They are typically bound to some kind of console, or standard output device. The streams after that are given no names, and can be used to implement file access or for custom streams.

The constant `_CONFIG_NFILE_STREAMS` found in `<nuttx/config.h>` defines the number of streams available.

15.5 Examine use of library

When working with a memory constrained system there is always a risk of running out of memory. Even when you are within the limits, it can still be good to know what is going on. To help with this the linker can produce a list file optionally with cross reference information. This is a valuable tool as it can tell you what has been placed in memory, where it is placed and why is it even there taking up 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, with:
  - What symbols are defined
  - What symbols are referenced
  - Who is referencing me
- An overall summary of total memory size

15.6 Library on a diet

If you are using the C library in a memory constrained system you may find that it quickly can consume code space. The reason for this is that the C library contains a decent amount of functionality and many functions acts a simple to use facades which actually covers a fair amount of code.

There are ways to tune things to reduce the memory foot print, but as always, you first need to understand what is going on and then consider what you want to do, if it becomes a problem.

15.6.1 Formatters

The C library contains different version of formatters which you can select depending on what capabilities you need. You can select a `printf()` formatter by specifying the variant you want on the command line to the linker using the `--rtattr` option:

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

The default print formatter is `reduced`. The following table describes the available print formatters:

Table 15.2: 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

Also the `scanf()` function exits in two variants, depending on if you want support for floating point numbers or not:

Table 15.3: 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

15.6.2 Reduced exit

Terminating the application includes code to close open files and run `atexit()` functions. Some applications, especially embedded applications are never meant to exit and having code to handle termination may be redundant. Also, closing any open files means that you need to have additional code related to files.

Also leaving `main()` means there is an implicit call the `exit()`. If having proper exit code is undesirable you may want to take actions to exclude it.

The simplest way to reduce the overhead of the exit code is to add `--rtattr exit=simplified` to the linker command line. This has the effect of not closing any open files or calling any `atext()` handlers. This will cause `exit()` to immediately jump to `_Stub_exit()` which is the name of the lowest level termination routine.

15.6.3 Consider alternatives

If library still occupy too much space, you may replace certain functions with smaller and less flexible alternatives.

15.7 System startup

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

The C startup module 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 module also contains a couple of small optional sections that are only included if needed. They are responsible for initializing optional parts of the run-time 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 module it can be found at `src/lib/lowlevel/cstartup.s` under the installation directory.

15.7.1 Customizing startup

In many cases the C startup module will probably work fine without any special 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 startup module, before any static C variables are given their start values.

If you need to make actual changes to the C startup module 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 module in the C library you need to provide a different value for the `cstartup` run-time attribute in your own `cstartup.s`. The `cstartup.s` assembly source file starts with:

```
;;; 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:

```
;;; 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 module, which is done using the `--rtattr` command line option:

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

Note: The C startup module has been carfully crafted to have a small footprint and to properly initialize the C run-time 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 run-time is needed in order to make non-trivial changes to the C startup module. Incorrectly made changes may cause linker errors, unused subsystem being pulled in, or result in a run-time that is not properly initialized.

15.8 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.

15.8.1 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.

CHAPTER 16

The optimizer

The compiler is capable of applying a range of optimizations which are aimed to reduce code size and increase performance.

It is beyond to scope of this chapter to go through all the optimizations that are applied. Instead it looks at how you can control optimizations and some aspects that may be useful to know.

16.1 Overview

By default the compiler performs basic rewrites, performs code selection that utilizes the instruction set well and attempts to make good use of the internal registers of the CPU.

The resulting application is in general well suited for debugging purposes. Many optimization passes are disabled by default.

16.1.1 General settings

You can enable optimizations on a broader scale 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.

16.2 Function inlining

Instead of making a call to a function, the body of a function can be expanded at the call site. This will typically increase performance, but may result in larger 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`.

16.2.1 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`.

16.2.2 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;
  }
}
```

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.

16.2.3 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 with `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.

16.2.4 Controlling inlining

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

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 functions 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.

16.3 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.

16.3.1 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 even 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.

CHAPTER 17

Assembly language interface

Assembly language is a symbolic way to access the actual machine instructions that the target machine is going to execute.

In certain situations you may want to control actual machine instructions being used. This can be because you need to use certain specfic instructions that cannot be expressed with normal C code. In some cases you may want very specific machine instruction sequences for interacting with hardware, interacting with exception stack frames, implement precise timing sequences or to implement a performance critical routine.

17.1 Intrinsic functions

The compiler provides some *intrinsic functions* declared in the intrinsics68000.h file. An intrinsic functions looks exactly like an ordinary function in use, but instead of making a function call, it generates a very specific sequence of instructions.

As an example, the intrinsic function `__disable_interrupts()` will emit machine instructions to disables normal interrupts. See *Intrinsic functions* for more information.

17.2 Assembly functions

The easiest way to access a routine written in assembly language is to implement a function in assembly language and call it like any other function from C. This requires that you follow the normal calling convention interface.

There is sometimes the thinking that it is better to use an inline assembler instead, but in reality they both have their pros and cons. The advantage of writing the code as an assembly function is that it provides a better separation between C and assembly which aids portability. The normal C calling convention is used. This specifies how to transfer values to the called function and handle return values.

There is a tiny amount of boiler plate assembly code to write, mainly to put the routine in a suitable section and to declare any public symbols. The assembly code is also put in a separate file which needs to be added to the build system.

Assembly language files usually have the file extension `.s` or `.asm`, but it varies wildly as assembly language itself is not standardized in any way.

The assembler provided by the Calypsi C compiler tool chain is modeled in a similar way as UNIX assemblers. Directives starts with a dot to avoid name clashes with instructions, which varies wildly between different targets. This allows a certain directive to be named the same way for a wide range of targets without any risk for naming conflicts.

As a bare minimum you need to 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 be able to call this function properly you need to provide a prototype on the C side:

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

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

17.3 Calling convention

The calling convention is fairly complex in all its details, but for most common situations it is reasonable simple.

If parameters are passed on the stack the caller is responsible for doing any cleanup. The called function can use any register resource, but it must obey that certain registers are to be preserved. If using registers that shall be preserved, the called function is responsible for saving the current value and then restore it before giving control back.

Parameters are passed in the `D0`, `D1`, `A0` and `A1` registers. These registers are destroyed by a function call. All other registers to be preserved by a function call.

Table 17.1: 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`

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 17.2: Return values

Register	Size	Types
`D0.B`	8	`char`
`D0.W`	16	`short`
`D0.L`	32	**`int`**, **`long`**, **`float`** data and function pointers

17.3.1 64 bits values

The above tables do not mention 64 bits values. A 64 bits value such as `long long` and `long double` is passed by reference, that is, as a pointer to the value. The size of this pointer is the same as the default pointer size and is either 16 bits or 24 bits depending on the data model used.

If the function returns a 64 bits value, the caller is responsible for allocating space for it, then add an extra invisible parameter to the function that holds a pointer to it. The called function is expected to return that pointer.

17.3.2 Structure passing

Structure parameters are passed on the stack. If the function returns a structure, the caller is responsible for allocating space for it, then add an extra invisible parameter to the function call. The called function is expected to return that pointer.

CHAPTER 18

Assembler

The assembler allows you to write assembly source files that can be used in C project. It is also possible to create pure assembly language projects.

18.1 Overview

Unlike Standard C there is no such thing as a 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 instruction set.

Expressions used in the assembly source are heavily inspired by C style operators and precedence rules.

The assembler will run the assembly source code through the C preprocessor which means you can use C comments, include files and do conditional inclusion in the same way as you would do in C.

Directives are mostly specific to the Calypsi C compiler tool chain and for the most part they look the same for all products provided by Calypsi. Directive names also start with a leading dot. This is common in many assemblers, but not universal.

Note: The main reason the dot prefix is adapted here is to avoid potential future conflicts in new products which could have a named mnemonic that could clash with a commonly used directive in the product line. Using a leading dot is a way to prevent future conflicts. It is also fairly common for assemblers to have a leading dot which makes the directive stand out in the source code.

18.2 Syntax

18.2.1 Source file format

Source lines in the assembly source follows the traditional style, with a label field starting in the first column, followed by a instruction that may take one or more operands. Comments are preceded by a semicolon that is mandatory:

```
[label[:]]   [instruction  [operands]]  [; comment]
```

The instruction can either be a mnemonic (name of a target instruction) or a directive. All directives start with a leading dot that is part of its name.

A label is a symbol that describes a location. It can be defined by placing it in the first column of a source line, optionally followed by a colon.

Labels can also be declared by importing it using the `.extern` directive.

Predefined words used in instructions (mnemonics and operands) are case *insensitive*, however symbols are case *sensitive*. Some examples of source lines follows:

```
; Assembler comments start with a semi-colon
;
Loop:         subq.l #1,d0     ; assembler comment
              nop
              BNE.S Loop
```

18.2.2 Symbol syntax

Symbols are case *sensitive* and can be of arbitrary length. A symbol starts with a letter or underscore and can be followed by letters, underscores and digits. Examples of symbols are `_4`, `a_symbol`, `abc` and `A1`.

It is permitted to use any character in a symbol, but in that case the symbol must be quoted using a back-tick. Such symbols can be `` `another symbol` `` and `` `Table: 5` ``.

The reason why predefined words are case insensitive is that both upper case and lower case assembly styles are wildly used.

Intermixing symbols with the same name but using different case is possible, but may be regard as bad style. The main reason why symbols are case sensitive in `as68k` is that C has case sensitive symbols and one of the duties of the assembler is to support the C compiler. It also forces mixed case symbols to be consistent in the assembly source code, which can be considered good style.

18.2.3 Preprocessor

The assembler uses a full featured C preprocessor to handle the input source file. The preprocessor provides the normal features you will find in a C preprocessor, the ability to include header files, macro expansions, conditional compilation and use of C style comments.

Wikipedia is a good place to look for an introduction with many examples on how to use the C preprocessor[1].

See the *Predefined macros* for available predefined macros, most of them are also relevant when running the assembler.

18.3 Sections

The assembly source file is divided into sections using the `.section` directive. A section is a unit of code or data that cannot be split up in smaller pieces. Sections are laid out in memory by the linker according to rules provided by a placement rules file.

[1] http://en.wikipedia.org/wiki/C_preprocessor

If you do not specify any `.section` directive the assembler pretends there has been a `.section code` in the source file. This means you will start off in a section named `code` unless told otherwise.

Using multiple sections allow more flexible placement of code and data by the linker compared to using a single section. A section name can be used multiple times and each use results in an individual section fragment.

All sections are relocatable and must be defined in the linker rules file.

18.3.1 Section kinds

The following table describes the different kinds of supported sections. If not specified, the section is assumed to be `text`.

Section kinds and modifiers are case insensitive.

Section kind	Description
`text`	Executable code.
`data`	An initialized data section in read/write memory (RAM).
`rodata`	An initialized data section in read only memory (ROM).
`bss`	Block Started by Symbol, intended to hold variables that are not given a value yet. A C compiler would normally zero fill such area before calling `main()`.

18.3.2 Section modifiers

A section modifier can be used to describe further behavior of the section. There are two such modifiers which can be specified either as positive or negative.

Section modifier	Description
`noreorder`	Obey the relative order between section fragments of the same name as encountered in a translation unit.
`reorder`	Allow section fragment to be placed in arbitrary order relative to other sections 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 refer to a label inside it. This is the default for library files.

Note: Section fragments with the same name an `noreorder` modifier are combined by the linker into a placement group that is placed as a single unit. The effect of `reorder` (or lack of `noreorder`) is that the section fragment is given its own placement group. This also have the effect that any following section fragments are combined in a new group separate from any section fragments before it. Thus, a section fragment with `reorder` not only causes it to be placed separately from the previous ones, it also makes a placement or `noreorder` sections before and after it separate placement groups.

18.3.3 Section alignment

The `.align` directive specifies a given alignment in address units. It ensures that the next location will have the specified alignment. This is done by advancing the location and inserting fillers if needed.

```
; Ensure that "table" label is placed at an address that can be
; evenly divided by 4.
        .section data
        ...
        .align   4
table:  ...
```

18.4 Expressions

Numeric expressions work in signed (2-complement) mode with a range check that depends on how the value is used. An error will result if the value exceeds its possible range. The *Operators* table shows the existing standard operators. In addition to these, there are also some specialized relocation and section operators, refer to *Relocation operators* and *Section operators* for more details.

You can use optionally use spaces between values and operators in an expression.

Table 18.1: 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

18.5 Numeric constants

Integer constants values can be entered in decimal, binary, octal or hexadecimal. A sequence of digits that do not start with a zero is considered to be a decimal integer value. Any sequence starting with `0x` is considered a hexadecimal integer value, a `0b` prefix is considered a binary integer value and any other sequence of digits starting with `0` is considered to be an octal integer value.

Character constants can also be used and they are replaced by their corresponding 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)
```

18.6 Location counter

The assembler converts the source program into machine code that can run on the 68000 processor. Each instruction will end up at some location in memory in consecutive order until the next `.section` directive (or end of file is reached). The address of the current instruction can be accessed by a single period (.), usually referred to as dot.

The dot label which contains the current instruction location is also called the *location counter* and it is incremented after each instruction so that it always contains the value of the start address of the current instruction.

The actual address value of the location counter is not known before the program is linked, but it can still be used in expressions. Short branches can be expressed using the location counter:

```
test:           tst.l   (10,a0)
                lda     (table),y
                bpl.s   .+4             ; skip next instruction of positive
                subq.l  #4,d0
```

However, it is often better to use labels or local labels instead, see below.

18.7 Local labels

Local labels start with a number and end with a dollar sign, i.e. `3$`. A local label is active between two non-local labels and cannot be exported to the linker. They are meant to be used as temporary locations for short distance branching, typically short skips or local loops.

```
 10$:            tst.l   (a0)+
                 beq.s   15$
                 subq.l  #1,d0
                 bne.s   10$
                 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 lables:

```
foo:             subq.l  #1,d2
                 bne.s   10$             ; error, 10$ 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.

18.8 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
`.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
`.rtmodel` *symbol, string*	Define a run-time 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

A *symbol-list* is a list of symbols separated by commas. An *expr-list* is a list of expressions separated by commas.

18.8.1 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 something else
; in the compilation unit is imported).
       .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.

.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 certain initialization code fragment exists, can request that such code fragment becomes active.

.rtmodel

It is possible to define run-time 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 run-time 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 essentially says that I know what I am doing and it will work with whatever value is in this attribute:

```
; I can work either season
              .rtmodel season, "*"
```

Functionally it 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 run-time attribute affects the entire compilation unit it appears in. If you need to have a more narrow scope for some run-time model attribute, you need to break up the source file into smaller pieces.

.byte

Defines one or more 8 bit values expressed as a list of comma separated expressions.

.word

Defines one or more 16 bit values expressed as a list of comma separated expressions.

.long

Defines one or more 32 bit values expressed as a list of comma separated expressions.

.quad

Defines 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

Takes 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.

.space

Takes 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

Emits 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.

.macro

Defines a macro, see *Macro language*.

18.9 Relocations

A *relocation* is an entity stored in the object file format that indicates a location in the generated code that needs to be altered by the linker. Relocations are generated automatically by the assembler and you do not normally need to think about them.

A relocation entry contains the location in the generated code, expression to be relocated and which kind of relocation to perform.

18.9.1 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. The `.word0` and `.word2` operators exists. The number refers to the byte offset they would stored at in memory. Thus `.word0` loads the upper 16 bits of value while `.word2` loads the low word.

```
; Take a 16 offset in some imagined 64K page
             move.w   #.word2 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.

`.word0`

The `.word0` operator gives the upper 16 bits word and is allowed to be resolved at link time.

`.word2`

The `.word2` operator gives the lower 16 bits word and is allowed to be resolved at link time.

`.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
```

18.9.2 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.

```
;;; 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 18.2: 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 of the given name. Different sections that are interleaved inside are silently included in the address range given by these operators.

18.10 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 `\`.

18.10.1 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, ignoring the order the parameters are given. 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.

18.10.2 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 in the code that uses it without any clashes:

```
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 3.5.1 #
#                                                       04/Jan/2022  21:36:00 #
# 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 203c0000               move.l  #100,d0
0006  00000004 0064
0007                                  waitreg d0
    \ 00000006 5380     1$:         subq.l  #1,d0
    \ 00000008 6600fffc             bne.w   1$
0008  0000000c 223c0000 1$:           move.l  #0,d1
0008  00000010 0000

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

Executable  (Text): 18 bytes
```

CHAPTER 19

Running the assembler

The interface to run the compiler is command line based. It is also possible to use the compiler from an IDE. In this case the IDE interfaces to the assembler using the command line.

19.1 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 an argument. The argument appears after option, they can be separated either by a space or an equal (=) sign. For single argument options, the argument is allowed to appear without any separator:

```
$ 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 `-v` or `--version`:

```
$ as68k --version
Calypsi assembler for Motorola 68000 version 3.5.1
```

Command line options are described in *Command line options*.

19.2 Include search path

Header file inclusion exists in two forms. You can either surround the filename with double quotes or angled brackets:

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

The angled form is normally used for system header files while the quoted form is used for application specific header files.

Note: There are no system header files provided in the installation for the the as68k assembler. The ability to have angled include files and searching the installation is retained as that is standard behavior of the preprocessor.

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

A header file specified in angle brackets are not searched relative to the compilation directory. Otherwise they behave identical.

The system header file directory is the installation directory.

Note: More precisely the system header file directory is a directory relative to actual cc68k executable file. This means that if you move an installation to another location in a the file system, it will still be able to find the correct system header file directory.

19.3 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.

19.3.1 Object file

The object file is in the ELF format with optional DWARF debugging information (if `--debug` (also named `-g`) option is specified).

The object file contains the following components:

- A symbol table
- Relocatable sections with code and data
- Relocations, to allow address values to be altered by the linker when the actual value is know after placing the section

- Source level debugging information in DWARF format (if the `--debug` command line option was specified)
- Vendor specific data, which carries certain additional information specific to the Calypsi C compiler tool chain. This includes runtime attributes as well as additional information not carried by ELF/DWARF, but that is used by the linker and debugger.

Note: DWARF debug information for the assembler consists of source line information. In addition the object file contains a symbol table.

19.3.2 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. Macro expansions are also shown. Finally there is a summary of the code and data sizes.

The following example is the `setjmp` source file from the C runtime library:

```
              .rtmodel version, "1"
              .rtmodel cpu, "*"


              .public setjmp
              .section code
              .align  2
setjmp:       move.l  sp,(a0)+
              move.l  a5,(a0)+
              move.l  (sp),(a0)+
              moveq.l #0,d0
              rts

              .section code
              .public longjmp
              .align  2
longjmp:      move.l  (a0)+,sp
              move.l  (a0)+,a5
              move.l  (a0)+,(sp)
              rts
```

If compiled with the `-l` option it will produce a list file that looks as follows:

```
###############################################################################
#                                                                             #
# Calypsi assembler for Motorola 68000                          version 3.5.1 #
#                                                        04/Jan/2022  21:36:00 #
# Command line: setjmp.s -l                                                   #
#                                                                             #
###############################################################################

0001                                    .rtmodel version, "1"
0002                                    .rtmodel cpu, "*"
0003
0004
0005                                    .public setjmp
```

(continues on next page)

(continued from previous page)

```
0006                                .section code
0007                                .align  2
0008  00000000 20cf     setjmp:       move.l  sp,(a0)+
0009  00000002 20cd                   move.l  a5,(a0)+
0010  00000004 20d7                   move.l  (sp),(a0)+
0011  00000006 7000                   moveq.l #0,d0
0012  00000008 4e75                   rts
0013
0014                                .section code
0015                                .public longjmp
0016                                .align  2
0017  00000000 2e58     longjmp:      move.l  (a0)+,sp
0018  00000002 2a58                   move.l  (a0)+,a5
0019  00000004 2e98                   move.l  (a0)+,(sp)
0020  00000006 4e75                   rts

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

Executable  (Text): 18 bytes
```

19.4 Command line options

This section covers the as68k command line options in detail.

19.4.1 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:

```
$ as68k
Missing: FILE

Usage: as68k [-v|--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
```

You can request a longer help using the --help option:

```
$ as68k --help
Calypsi assembler for Motorola 68000 version 3.5.1

Usage: as68k [-v|--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]
```

(continues on next page)

(continued from previous page)

```
             [--data-model NAME] FILE
  use 'as68k --help' for detailed help

Available options:
  -v,--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
  -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 run-time attribute (identifier or quoted
                           string value accepted)
  --weak-symbols           Make all public symbols entries weak
  --core CORE              CORE, one of '68000' or '68010' (defaults to '68000')
  --target TARGET          Target system, one of 'Amiga' or 'Foenix' (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
```

19.4.2 Options in detail

`--version`, `-v`

Displays the name and version of the compiler.

`--output-file`, `-o`

Use this option to specify the name of the output object file. If not given, the output file is created from the name of the input file (ignoring any directory path) and sets the file extension to `.o`. Thus, output files are written to the compilation directory by default.

You can use this option to alter the name of the output file as well as providing a directory path to it. Such 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`.

The `-l` and `--list-file` options are mutually exclusive, only one of them can be stated on the command line.

`--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.

The `-l` and `--list-file` options are mutually exclusive, only one of them can be stated on the command line.

-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.

-D

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

-U

Undefine a preprocessor symbol. 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.

-g

Synonym for `--debug`.

--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 or to select a particular variant of a function that exists in many variants.

--weak-symbols

Make all public symbols in the object file weak. This is normally not needed, but can be used in certain situation to make a default implementation of some function in a library. This makes it possible to override the function with one that is not weak, or fall back to use the default one when no replacement is provided.

--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 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 20

Linker

The linker is used to combine object files produced by the C compiler and assembler into an executable.

Object files can also come from *libraries*. One example of this is the supplied C runtime library.

The linker step for a cross compiler requires more information than what is given to a typical hosted linker. This is because the target system where the application will run needs to be described as it can look very different for different targets.

20.1 Running the linker

The interface to run the compiler is command line based. It is also possible to use the compiler from an IDE. In this case the IDE interfaces to the linker using the command line.

20.1.1 Basic invocation

The linker is invoked in the following way:

```
$ ln68k [options] [object-files] [library-files] rules-file
```

The linker requires a rules file to work. This file uses the filename extension `.scm` and contains information about the memory of the target system and section placement rules.

Command line options and input files in the command line can appear in any order.

Command line *options* are optional arguments that tune the behavior of the linker. They always start with a dash character. There are two variants, single letter options (`-l` to instruct the linker 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 an argument. The argument appears after option, they can be separated either by a space or an equal (=) sign. For single argument options, the argument is allowed to appear without any 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 `-v` or `--version`:

```
$ ln68k --version
Calypsi linker for Motorola 68000 version 3.5.1
```

20.2 Memory

When describing the target system the linker uses a concept of memories. A *memory* is a named entity that specifies a continuous range of storage locations. A description of a memory have the following attributes:

name The name of the memory. The memory name is the identifier used for a particular memory area

address range The start and end address (inclusive) specify the address range of the memory.

section Here you put the section names you want to bind to the current memory.

placement group This is a group of sections that are placed together in a cloned version of the parent memory. This allows a memory range (the parent memory) to contain more than one memory with different type properties.

fill word Value used to fill unused locations in the memory. Defaults to zero, but can specified as `(fill 0)`.

20.3 Linker rules file

The rules file (extension `.scm`) describes how the section fragments are to be placed in memory. This file serves three purposes, *a*) it defines the actual memories; *b*) it defines which sections to expect; and *c*) it describes where the sections can be placed in memory.

20.3.1 Memory rule

A memory rule defines a memory. It also defines which sections (by name) that will go into that memory. There can be multiple memory rules in a rule file and you will need one for each memory block.

A 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, using them makes the file easier to read. However, the use of characters like the single quote, period and parentheses are important to make the linker tool able to parse the file correctly.

20.3.2 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.

The `nearcode` section has an address range, which should be a sub-range of the memory range. The mentioned section can be placed anywhere in this sub-range.

Fixed placement

The `reset` section is placed at the fixed address `0x00000000` by having only a given address, not a range.

20.3.3 Block rule

Normally sections are created by the compiler or 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 defining 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))
```

20.3.4 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 is 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))
```

(continues on next page)

(continued from previous page)

```
    (block sstack (size #x200))
   ))
```

Here the user stack size is set to `1000` hexadecimal (4096 bytes decimal) and is bound to a 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: It is possible to override the stack size defined in the linker rules file by using the command line option `--stack-size`.

20.3.5 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 semi-colon is a comment.

Note: It is possible to override the heap size defined in the linker rules file by using the command line option `--heap-size`.

20.3.6 Base address

A *base address* area is a region of memory that is permanently pointed to by some register in the 68000 CPU. Using base addressing to access objects is typically more efficient, as it makes use of a more efficient addressing mode on the 68000.

A base address is defined in the linker rules file using a `base-address` rule. It works by defining a special symbol that corresponds to the base address area and tie it to a memory that is the actual area covered by the base address.

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 bits 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.

20.3.7 Placement groups

A *placement group* is a named group of sections inside a memory that share a common property, such as that they define actual value bits or represent a BSS area. In many output formats such areas need to be separated. However, it can make sense that a memory that covers a certain address range can have sections with different such properties.

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 there is a `near-bank` memory that contains two placement groups `near-bits` and `near-nobits`. The `near-bits` contains two sections `near` and `cnear` that define actual value bits forthe final application. 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 an embedded system they should go into completely different memories.

Note: The names `near-bits` and `near-nobits` have no special meaning to the linker.

20.4 Linker list files

A list file can be generated from the linker using the `--list-file` (or `-l`) command line option. The list file can also contain a cross reference by specifying the `--cross-reference` 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 <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
-------------------------------------------------------------------------
NearRAM 00200000-0020ffff 00010000   0.0%   none      00010000
RAM     00210000-0023ffff 00030000  10.7%   none      0002ae00
Vector  00000000-000003ff 00000400   0.8%   none      000003f8
flash   00100000-001fffff 00100000   0.0%   none      000fffb2


####################
#                  #
# Sections summary #
#                  #
####################

Name   Range              Size     Memory Fragments
---------------------------------------------------
heap   00210000-00213fff  00004000 RAM    1
stack  00214000-00214fff  00001000 RAM    1
sstack 00215000-002151ff  00000200 RAM    1
reset  00000000-00000007  00000008 Vector 1
code   00100000-0010004d  0000004e flash  7


################
#              #
# 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
          >  code  0000002e
          >  reset 00000008
  4  simplified_exit.o clib-68000-lc-sd.a
          # picked based on exit=simplified
          >  code 00000008
  5  debug_exit.o      clib-68000-lc-sd.a
          # picked based on semiHostedStub=1
          >  code 00000012
  6  debug_break.o     clib-68000-lc-sd.a
          # picked based on semiHostedStub=1
          >  code 00000002

###################
#                 #
# Cross reference #
#                 #
###################

Section 'heap'  placed at address 00210000-00213fff of size 00004000
(linker generated)

Section 'stack'  placed at address 00214000-00214fff of size 00001000
(linker generated)

Section 'sstack'  placed at address 00215000-002151ff of size 00000200
(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 4 section index 2)
        main in (main.o unit 0 section index 2)
```

(continues on next page)

(continued from previous page)

```
_Stub_exit in section 'code'
 placed at address 0010002a-0010003b of size 00000012
(debug_exit.o (from clib-68000-lc-sd.a) unit 5 section index 2)
    Defines:
        _Stub_exit = 0010002a
    References:
        _DebugBreak in (debug_break.o (from clib-68000-lc-sd.a) unit 6 section index 2)
    Referenced from:
        exit (simplified_exit.o (from clib-68000-lc-sd.a) unit 4 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 4 section index 2)
    Defines:
        exit = 0010003c
    References:
        _Stub_exit in (debug_exit.o (from clib-68000-lc-sd.a) unit 5 section index 2)
    Referenced from:
        (cstartup.o (from clib-68000-lc-sd.a) unit 2 section index 6)

main in section 'code'  placed at address 00100044-00100047 of size 00000004
(main.o unit 0 section index 2)
    Defines:
        main = 00100044
    Referenced from:
        (cstartup.o (from clib-68000-lc-sd.a) unit 2 section index 6)

__low_level_init in section 'code'
 placed at address 00100048-0010004b of size 00000004
(cstartup.o (from clib-68000-lc-sd.a) unit 2 section index 7)
    Defines:
        __low_level_init = 00100048
    Referenced from:
        __program_start (cstartup.o (from clib-68000-lc-sd.a) unit 2 section index 2)

_DebugBreak in section 'code'
 placed at address 0010004c-0010004d of size 00000002
(debug_break.o (from clib-68000-lc-sd.a) unit 6 section index 2)
    Defines:
        _DebugBreak = 0010004c
    Referenced from:
        _Stub_exit (debug_exit.o (from clib-68000-lc-sd.a) unit 5 section index 2)

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

Executable       (Text):    86 bytes
Non-initialized        : 20992 bytes
```

The linker list file that can be optionally generated shows how the section fragments are distributed in the memories and the total size of memory allocated.

The `Memories summary` part summarizes the different memories used in the application. Here you will see the size of the memory, how much memory is in use and the largest unallocated continuous memory block. The checksum column lists the calculated checksum, which may also be included in the image.

The `Sections summary` part gives an idea of where the sections are located and how many section fragments there are for each one.

The `Object files` part shows the object files and what library files they were taken from (if any). It also shows the sections and overall size contribution.

The `Cross reference` part is included in the list file if `--cross-reference` is specified on the command line. For each section it shows the entry point, the section name, the address range it is placed at and the size. It then lists defined symbols, what symbols are referenced in other sections and who is referencing this section. This means you will see full reference information in both directions for every section included in the application.

Finally the `Memory sizes` summarizes the total amount of memory in use. Note that the amount of memory is given in decimal here, otherwise hexadecimal numbers are used in the list file.

20.5 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.

20.5.1 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`.

20.5.2 Intel hex

This is the Intel hex file format which contains the output binary expressed in ASCII text form. The output file has file extension `.hex`.

20.5.3 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`.

20.5.4 Raw

The `raw` format is the program memory as a raw image file. The output file has file extension `.raw`.

!linkerOutputFormats

20.6 More on linker rules

The linker rules file is actually a source file for the Scheme programming language, which is the reason for the choice of the `.scm` file extension.

You do not need to be familiar with Scheme to specify linker rules, simply follow the examples to set things up.

The shown examples are just a Scheme program that consists of a single variable named `memories` which is bound to a data structure. The linker runs a Scheme interpreter to read the file and then look for the resulting `memories` variable and peek into its contents. This has two implications, *a*) the syntax of the file is dictated by Scheme, which is based on s-expressions[2]; and *b*) it is possible to use a more elaborate program with Scheme macros to generate the memory rules.

20.7 Command line options

This section covers the `ln68k` command line options in detail.

20.7.1 Options overview

If you run the compiler from the command line without any arguments it will complain that it needs some object files to work with:

```
$ ln68k
the linker requires some object files to work with (use --help for help)
Terminating due to errors
```

You can request a longer help using the `--help` option:

```
$ ln68k --help
Calypsi linker for Motorola 68000 version 3.5.1

Usage: ln68k [-v|--version] [-o|--output-file OUTPUT-FILE] [-g|--debug]
             [--semi-hosted] [--no-data-init-table-section]
             [--override IDENTIFIER] [--cross-reference] [--rtattr NAME=VALUE]
             [--verbose] [-l] [--list-file LIST-FILE]
             [--memories-expression EXPRESSION] [--program-root SYMBOL]
             [--program-start SYMBOL] [--copy-initialize SECTION]
             [--no-copy-initialize SECTION] [--output-format FORMAT]
             [--stack-size SIZE] [--heap-size SIZE] ([--hosted] | [--rom-code])
             [--core CORE] [--target TARGET] [FILE...]
  use 'ln68k --help' for detailed help

Available options:
  -v,--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
                           useful for assembly projects)
```

(continues on next page)

[2] https://www.wikipedia.org/wiki/S-expression

(continued from previous page)

```
  --override IDENTIFIER    Override symbol in archive library (treat it as weak)
  --cross-reference        Include cross reference and map information in the
                           list file
  --rtattr NAME=VALUE      Specify run-time 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
  --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'
  --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
  --output-format FORMAT   Format, one of 'intel-hex', 's-record', 'raw' or
                           'pgz'
  --stack-size SIZE        Stack size override (size normally 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' or '68010' (defaults to '68000')
  --target TARGET          Target system, one of 'Amiga' or 'Foenix' (defaults
                           to embedded/ROM use, if omitted)
  -h,--help                Show this help text
```

20.7.2 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 routine in the library with your own.

--cross-reference

Also include cross reference information in the list file.

--rtattr NAME=VALUE

Use the indicated function when multiple alternatives exists in the library. This can be used to select the capability of the `printf()` formatter, `scanf()` and `exit()`.

You may want to select the smallest variant that has enough functionality for you application.

--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`.

The `-l` and `--list-file` options are mutually exclusive, only one of them can be stated on the command line.

--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.

The `-l` and `--list-file` options are mutually exclusive, only one of them can be stated on the command line.

--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.

--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 additional an additional file format of the produced executable application. An ELF executable is always produced.

--heap-size

The heap size is normally 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 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.

--stack-size

The stack size is normally 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 that you run on some kind of hosted environment. This will cause data object initialization by copying from ROM to be omitted. Instead it is assumed that the application is loaded into memory and initialized data objects gets their initializer values by 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 the 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 and 68010 are supported.

`--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.

CHAPTER 21

Librarian

With the `nlib` library tool you can build *libraries* (sometimes called *archives*) which is a single file that contains multiple object files and a symbol index.

21.1 Creating a library

To create a library, simply 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 normally used as input to the `ln68k` linker by just specifying it on the command line. When linking with a library, only those section fragments that are actually used are included in the final output.

21.2 File format

The file format used is the same as `ar` on UNIX System V and GNU. The created archive contains an index to make it faster to select the objects that are needed from the archive.

21.3 Command line options

The only command line options supported are `--version` and `--help`.

To display the version of the linker, use `-v` or `--version`:

```
$ ln68k --version
Calypsi linker for Motorola 68000 version 3.5.1
```

CHAPTER 22

Debugger introduction

The Calypsi tool chain debugger is a source code debugger allows you to see what happens inside programs while it executes.

The debugger provides both interactive as well as script based control.

The design of the debugger is based on ideas coming from command line debuggers like GDB (GNU project debugger) and LLDB (LLVM project debugger). If you are familiar with these you will find many similarities.

22.1 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 repeat the last command when pressing the return key, you can do the many common actions and repetitive commands using single or a few key strokes.

The debugger can also be used with several debugger front ends, such as Visual Studio Code, Eclipse and Emacs. This is possible thanks to the MI (Machine Interface) protocol designed by the GNU project for this purpose.

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. This is made possible by that the complete debugger functional API is provided to Lua. The debugger Lua API even goes further and provides additional concepts, such as making it possible to define new console commands and subscribe to notifications on internal debugger events.

In other words, with the Calypsi debugger you get the best from both the command line world and integrated graphical development environments.

22.2 Running the debugger

The interface to run the compiler is command line based and you can either use it from a terminal or connect it to an IDE.

22.2.1 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 by stating just its name. This will put you in an interactive debugger session:

```
$ db68k
```

Normally you want to 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 the behavior of the debugger. They always start with a dash character. There are two variants, single letter options starts with a single dash. The other variant is long descriptive options that starts with two dashes.

To display the version of the debugger, use `-v` or `--version`:

```
$ db68k --version
Calypsi debugger for Motorola 68000 version 3.5.1
```

22.3 Command line options

This section covers the `db68k` command line options in detail.

22.3.1 Options overview

Use the `--help` option to display the available command line options.

```
$ db68k --help
Calypsi debugger for Motorola 68000 version 3.5.1

Usage: db68k [-v|--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] [--program-start SYMBOL] [--target-remote DEVICE]
             [FILE]
  use 'db68k --help' for detailed help

Available options:
```

(continues on next page)

(continued from previous page)

```
-v,--version             Display version number
-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' or '68010' (defaults to '68000')
--target TARGET          Target system, one of 'Amiga' or 'Foenix' (defaults
                         to embedded/ROM use, if omitted)
--program-start SYMBOL   Program start symbol, defaults to '__program_start'
--target-remote DEVICE   Use remote device to communicate with target
-h,--help                Show this help text
```

22.3.2 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, this is also the default. The machine interface interpreter is used when the debugger is controlled by an IDE.

`--eval-command`, `-e`

This specifies a command that is executed before giving control to the interactive session. This option can appear several times on the command line to allow multiple commands to be specified.

–batch

Terminate the debugger session after all commands specified by `--eval-command` have been processed. This is useful when running the debugger in an automated way from some script.

–tty, -t

Specifies the console `tty` for the application. If the applications writes to or read from the standard stream, this provides the `tty` device where it is done.

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.

–logging-enable

Enable logging when debugger starts.

–logging-to-file

Specifies the file to use for logging output.

–logging-overwrite

Specifies whether logging to the file should append to the file or write over any previously existing log file.

–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 program starts, take control immediately and stop the program. This allows you to debug a program from the very beginning (before any `main()` entry is reached) without having to figure out where to set the breakpoint to do that.

–terminate-on-program-exit

If the application exists, also exit the debugger session. The default is that the debugger notifies that the application has exited and stays in the debugger session.

–exit-breakpoint

Insert a breakpoint at the `exit()` function. If you link with debug stubs you do not need to specify this option as handling of exit is provided by the debug stubs.

`--target-remote`

Specifies the communications device to be used to connect to a target for debugging. This disables the built-in simulator and configures the debugger for remote debugging using the gdbserver protocol which allows for connecting to a remote target, typically used for debugging on real hardware.

`--program-start`

This symbol defines the first actual executable instruction in the program. If you are building a normal C project you should not change this.

`--core`

Specifies the core used.

`--target`

Specifies a certain target system.

CHAPTER 23

Console command line

The console interpreter is intended for interactive command line use. In this mode you have access to a powerful command line interpreter with command completion, command history and command line editing.

Your command line history is also saved between sessions in `~/.db68k/db68k.history`.

23.1 Command structure

Commands are made up from one or more space separated words. If a given word is long enough to be a unique match, it does not need to spelled out in full.

After the command there may be an argument that is specific to the command, e.g. a string, file path or boolean value.

23.1.1 Command completion

Command completion is available using the `TAB` key. Pressing `TAB` directly at the prompt provides a list of all words that can be used to 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 will complete the command to the left of the cursor as far as possible. If the word can be partially completed the first TAB will fill in as far as possible:

```
(db68k) int<TAB>
```

Results in:

```
(db68k) inter
```

Pressing TAB a second time results in:

```
(db68k) inter<TAB>
interpreter-exec  interrupt
(db68k) inter
```

23.2 A simple 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 only need entered with enough characters to make it unique. For the command `info breakpoints` it is enough to type `i b` followed by return. This works because the command interpreter takes all words in account. The command line completer looks at what is to the left of the cursor, so if you type `i<TAB>` you will get all command alternatives that start `i`. To make it complete to `info` you need to type `inf<TAB>` followed by `b<TAB>`. However, if you type `i b<TAB>` it will expand to `i breakpoints`.

To debug the program you need to `run` (which can be entered as just `r`) it which starts execution from the beginning. The program will stop almost immediately as it hits 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 the source file `main.c:16`. Some 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)
```

You use the step command that can be entered as s. This steps one source line. The second step you simply press RETURN which repeats the previous command (s in this case).

Each step moves us one line further ahead and the source context shown moves along with it.

The step command steps into functions which can be seen as you entered basic_expr().

You can inspect variables with info locals.

```
(db68k) info local
(int) a3 = 3
(long) a2 = 2
(long) a = 21
(db68k)
```

You can see how you got to the current position use backtrace which can also be entered as bt.

```
(db68k) backtrace
#0 0x8000 in basic_expr() at basic_expr.c:14
#1 0x8137 in main() at main.c:17
(db68k)
```

Looking at the source code, you realize that you want to stop next at a certain line a bit further down, this can be done as follows:

```
(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) {
```

(continues on next page)

(continued from previous page)

```
   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
0x8030 in basic_expr() at basic_expr.c:18
(db68k)
```

The `continue` command (short form `c`) is used to resume execution (not `run` which runs the program from start).

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)
```

This gives two additional variables `b` and `cglob_int` which have been added to the local context. At this point the variable `a` does not have a value. Looking at the source code you can see that it is going to get a new value at this line and its last use was two lines up, at line 16. This means the debugger knows about `a`, but at this location it does not hold a value that can be examined.

Step over two function calls and inspect the value of `a` using the `print` command (which can be entered as just `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)
```

Now `a` has a value from the function calls.

You can see in the code that `cglob_int` is a local variable of type `char`. It is actually shadowing a global variable with the same name that is brought into scope at line 23. If you step to line 24 the global `cglob_int` should be 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 visible again, with a different value and type.

You realize that you do not need to use the first breakpoint anymore. It can be removed using `delete`, but you decide to to disable it using the `disable` command instead:

```
(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)
```

Here you can see that the first breakpoint still exists, but it is not enabled (`n` in the `Enb` column). You can also see that the debugger keeps track of some statistics about the breakpoints.

CHAPTER 24

Remote debugging

This chapter describes how to set up debugging to a remote target. In such scenario the debugger runs on the host machine and connects to and control an application being debugged on actual hardware.

The communications protocol used is the gdbserver protocol which can be implemented as either a small agent or by a bridge process on a host machine which translates from the gdbsever protocol to the protocol supported by the hardware.

24.1 Serial port

A simple way to implement target debug support is to put a small debug agent software on the target. The ubiquitous RS-232 serial port can be used in this case. As typical host computers are no longer equipped with such ports, you can use a USB to serial converter, such as the FTDI USB-RS232 which typcially comes with a bare wire end that needs to have a suitable connector soldered on.

Most major operating systems are already equipped with drivers for the FTDI USB-RS232 cable, but if you choose other cables you may need to install a driver. You may need to ask the vendor of the USB-RS232 cable for a suitable driver.

24.1.1 Finding the device

On Linux and macOS you can typically find the serial port device in `/dev/ttyUSBn`, `/dev/ttySn` or similar. You can use the `dmesg` command to query about 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 (number 2 and 3) wired the opposite way commpared to the DE-9 connector.

24.1.2 Setting the speed

The speed and serial port settings need to be done outside the Calypsi db68k debugger and the settings depends on how your debug agent is configured.

On Linux and macOS you can use the `stty` command to set up the serial port:

```
$ stty 115200 -F /dev/ttyUSB1
```

If you have the `screen` utility program installed it can be started with parameters to configure the serial port:

```
$ screen /dev/ttyUSB1 115200
```

This will leave the serial port configured when you leave the `screen` application.

You can leave it by pressing control-A followed by a backslash (`c-a \`).

Hint: One benefit of using `screen` is that it can be used to see whether you are properly connected at a matching speed with the debugger agent. Simply press a key and you should see a packet reply `$S13#b7`.

24.1.3 Installing the agent

The Calypsi remote debugger agent for the serial port needs to be downloaded, built and executed on the target hardware. You can download it from https://github.com/hth313/Calypsi-remote-debug.

24.1.4 Starting the debugger

Once the debugger agent is running on the hardware and the communications channel has been configured you can start the Calypsi db68k debugger using:

```
$ db68k --target-remote /dev/ttyUSB1 application.elf
```

CHAPTER 25

Visual Studio Code

Visual Studio Code[3] is a lightweight but powerful source code editor which runs on multiple platforms.

With the C/C++ for Visual Studio Code extension[4] it is possible to use the Calypsi C compiler tool chain in Visual Studio Code.

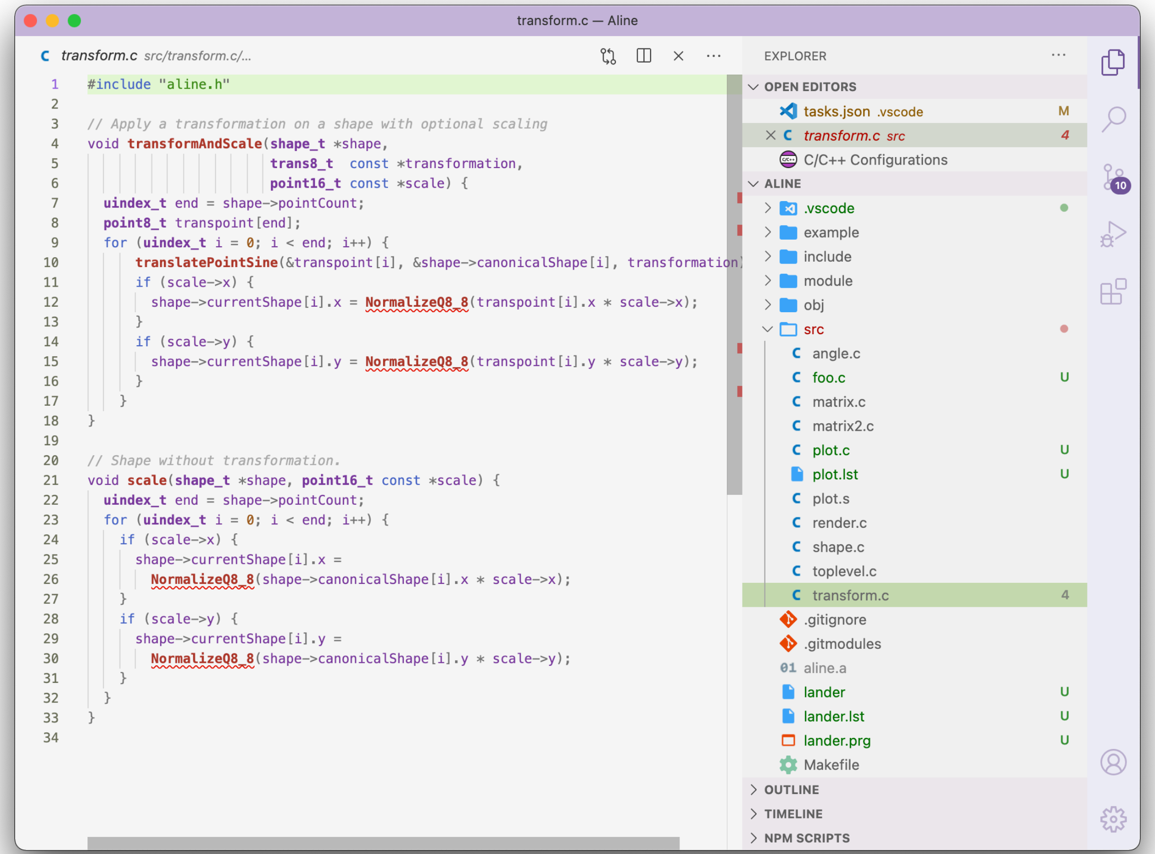
This works as Visual Studio Code can be tailored to use different build and debugger tools. The most crucial part is perhaps the debugger integration which is based on the MI (Machine Interface), which is supported by the Calypsi db68k debugger.

Visual Studio Code is a good choice for a graphical user interface to Calypsi as it is very actively developed, lightweight, responsive and highly configurable.

Documenting the full Visual Studio Code is beyond the scope of this guide. Refer to the official Visual Studio Code documentation[5] for detailed information.

Some basics on how to get started is covered here as it may not work entirely as you may expect from traditional IDEs.

[3] https://code.visualstudio.com/
[4] https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools/
[5] https://code.visualstudio.com/docs



```c
#include "aline.h"

// Apply a transformation on a shape with optional scaling
void transformAndScale(shape_t *shape,
                       trans8_t  const *transformation,
                       point16_t const *scale) {
  uindex_t end = shape->pointCount;
  point8_t transpoint[end];
  for (uindex_t i = 0; i < end; i++) {
    translatePointSine(&transpoint[i], &shape->canonicalShape[i], transformation)
    if (scale->x) {
      shape->currentShape[i].x = NormalizeQ8_8(transpoint[i].x * scale->x);
    }
    if (scale->y) {
      shape->currentShape[i].y = NormalizeQ8_8(transpoint[i].y * scale->y);
    }
  }
}

// Shape without transformation.
void scale(shape_t *shape, point16_t const *scale) {
  uindex_t end = shape->pointCount;
  for (uindex_t i = 0; i < end; i++) {
    if (scale->x) {
      shape->currentShape[i].x =
        NormalizeQ8_8(shape->canonicalShape[i].x * scale->x);
    }
    if (scale->y) {
      shape->currentShape[i].y =
        NormalizeQ8_8(shape->canonicalShape[i].y * scale->y);
    }
  }
}
```

25.1 Installation

Installation is easy to do by downloading it from the official site. After that you can easily update from inside Visual Studio Code itself.

Note: If you are using Arch Linux or a distribution based on it like Manjaro, you can install VS Code using its package system. At the moment the community build package is named code, but due to licensing issues it does not allow installing the vscode-cpptools extension. To make it work you need to install (the Microsoft-branded installation) `visual-studio-code-bin` from the AUR.

25.2 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.

If you have an existing project, you can just select `File>Open` and browse to the top folder in that project, the press `Open` and your project is loaded.

Project specific files are kept in an `.vscode` directory in your project directory. Initially there are no such files.

25.3 Building

Once you have populated your project and perhaps created a `Makefile`, you can add a task to build it. Select `Tasks>Configure Tasks...` and pick `Create tasks.json file from template`. Then select `Others` to get a `tasks.json` that you can edit. It may end up looking something like:

```
{
    "version": "0.2.0",
    "tasks": [
        {
            "label": "build",
            "type": "shell",
            "command": "make -k",
            "options": {
                "cwd": "${workspaceRoot}/src"
            },
            "problemMatcher": [
                "$gcc"
            ]
        },
        {
        "label": "clean",
        "type": "shell",
        "command": "make -k clean",
        "problemMatcher": [],
        }
    ]
}
```

25.4 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.

25.4.1 Configuration

Before you can start the debugger you need to 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",
```

(continues on next page)

(continued from previous page)

```
            "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`[6]

The launch configuration is a text file that is stored in `.vscode/launch.json`.

25.4.2 Running the application

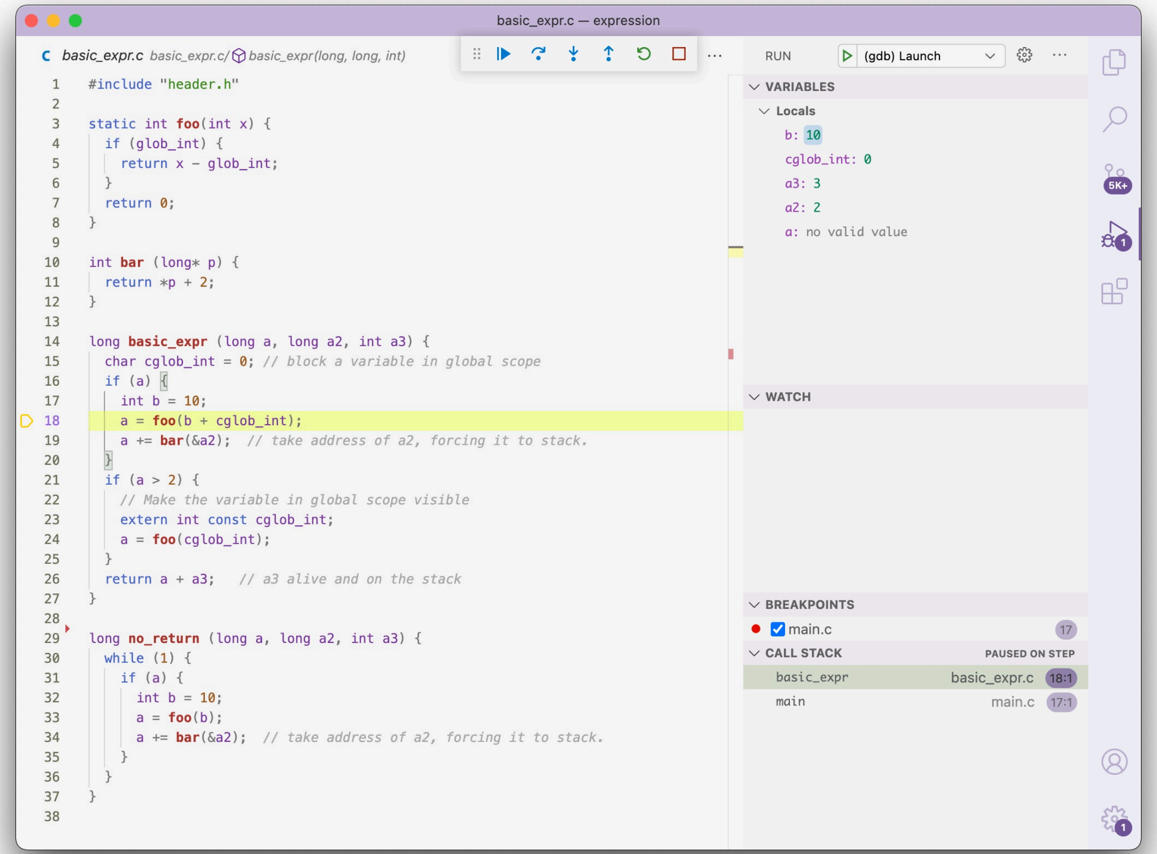
Once configured you can run your application using `Run >> Start Debugging` which is bound to function key 5 (`F5`).

You may want to insert a breakpoint in your application before you start, or you can take control by interrupting it once started.

If the debugger windows to show variables, call stack and other information do not show, you can click on the little bug insect by the side to switch to the debugger view.

[6] 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. As the application being debugged is not a UNIX process at all, it does not work. Instead you need to use the `lldb` mode, where the MI command `-exec-interrupt` works properly, just as it does with `db68k`.

Reference https://sourceware.org/bugzilla/show_bug.cgi?id=20035


basic_expr.c — expression
basic_expr.c basic_expr.c/basic_expr(long, long, int)
RUN
(gdb) Launch

```
#include "header.h"

static int foo(int x) {
  if (glob_int) {
    return x - glob_int;
  }
  return 0;
}

int bar (long* p) {
  return *p + 2;
}

long basic_expr (long a, long a2, int a3) {
  char cglob_int = 0; // block a variable in global scope
  if (a) {
    int b = 10;
    a = foo(b + cglob_int);
    a += bar(&a2);  // take address of a2, forcing it to stack.
  }
  if (a > 2) {
    // Make the variable in global scope visible
    extern int const cglob_int;
    a = foo(cglob_int);
  }
  return a + a3;   // a3 alive and on the stack
}

long no_return (long a, long a2, int a3) {
  while (1) {
    if (a) {
      int b = 10;
      a = foo(b);
      a += bar(&a2);  // take address of a2, forcing it to stack.
    }
  }
}
```

VARIABLES

Locals

```
b: 10
cglob_int: 0
a3: 3
a2: 2
a: no valid value
```

WATCH

BREAKPOINTS

main.c	17

CALL STACK — PAUSED ON STEP

basic_expr	basic_expr.c	18:1
main	main.c	17:1

CHAPTER 26

Lua scripting

The debugger can be controlled with the means of the Lua language[7]. Scripting allows you to control and extend the debugger by the means of writing programs that are executed inside the debugger. In its simplest form, it allows you to write scripts (basically corresponding to batch file under Windows). More advanced use allows you to add your own commands to the existing command set. Plug-in like behavior allows you to have very tight integration with the debugger, making it possible to listen to internal events, control execution and debugger behavior.

From a more high level perspective, Lua together with the debugger makes it possible to do a lot of different tasks:

- Perform setup tasks to prepare your debug session.
- Simplifying repetitive tasks.
- Running fully automated tests.
- Extend the functionality, for example to provide introspection of the state and behavior of an operating system or a protocol stack. Such extended commands are simply added to the command set and share the same command line interface as any built-in command.

You can use any kind of automation level. Run the whole session under the control of a script, or provide a set of high level functionality using commands.

26.1 Running a script

To get started, this simple Lua program just displays the registers:

```lua
local db = require('db')

print(db.consoleCommand('info registers'))
```

If saved in `script-name.lua` (in current directory), you can run the script with:

[7] https://www.lua.org/

```
(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`.

Note: Use the `--eval-command` (can also be used as `-e`) to invoke a Lua script from the command line, 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.

26.2 Lua environments

Each time you run a script, a new fresh Lua state is created. This means you will start out with standard and debugger libraries available for import, 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 your script provides functions to the debugger, for example, by creating new console commands, the Lua state is as part of the command description. Invoking such command later will result in that the *same* state it was created in is used again. In such case, everything is retained, like global variables and closures. This allows commands to work in a state that has been set up for it, and it allows the command to depend on other things that has happened in this state before, like previous uses of the same (and other) commands created by the same Lua script.

A plug-in typically register one or more Lua functions with the debugger to be called under certain circumstances. Such functions always execute in same state as they were installed in.

It can be good practice to provide a certain set of related functionality, like a command set and plug-in behavior in a script of its own. Multiple such plug-ins can be installed, each by its own `source` invocation, allowing a plug-in to have its private state shared, while allowing multiple plug-ins to co-exist separated from each other.

26.3 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 is also provided.

26.4 Console commands

The db table makes it possible to send console command to the debugger. This will execute a command just as it is typed at the terminal. The result of the command is passed back as a string to Lua.

If you want to do something with the result other than displaying it (or perhaps just ignoring it), you will need to parse it. Writing such parsers quickly becomes tedious as the text output is really just intended to be displayed.

26.5 MI commands

MI commands provide a more precise command interaction intended for debugger front-ends. These commands are in general a much better fit for use from a Lua script. While the commands are still given as strings, the result is converted to nested Lua tables with basic Lua types such as boolean, numbers and strings.

This greatly simplifies interpreting the result of a call made from Lua to the debugger.

26.6 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. Return values are just the value returned or a failure, instead of the MI style result hierarchy.

26.7 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.

26.8 Arrays and Lua

Lua differs from C in that an array index starts at 1. This mismatch between the two languages has some consequences that can worth knowing when dealing with debugger C expressions from Lua.

To make it easier to work with C expressions you can use the behavior of C and start array results with 0 also in Lua. The *ipairs* iterator function in Lua assumes (naturally) that you are a Lua programmer and start arrays with 1 which means that iterations of array elements this way will skip the element at index 0.

The alternative would be to shift the index in expressions, but that would probably be even more confusing and error-prone. You are after all dealing with C expressions.

26.9 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 27

Lua data types reference

In this section we go through the various data structures that are used. Anything more complicated than a boolean, number or string, are represented by a table.

27.1 Address

Internally the debugger uses addresses which consists of triples. An address has a numeric value, a memory type and optionally a bank number. Addresses are represented in Lua by a table with the following fields:

Table 27.1: 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.

27.2 Adapted MI functions

Many functions provided in Lua are actually adapted from the set of *machine interface* (MI) commands, which is the commands that are passed between a debugger front end (such as Eclipse or VS Code) and the debugger.

The functions are available in the `db` table and the Lua interface takes care of translating the arguments for the function and passing it on to the debugger command handler.

Results are passed back and are translated to Lua. As the underlying mechanism is a command protocol, the result table is a little bit more elaborate than what first be guessed.

At top level a result have the following fields:

field	type	description
class	string	The result class, for many commands this will be done. An error is given as `error`. A complete list is provided below.
value	integer	This is the actual result, which depends on the actual function called.
type	string	The type of the reply. For function calls, this will be `result`.

CHAPTER 28

Lua function reference

In this section we cover the functions provided by the debugger module. They are grouped in families, so that you will find commands that are related together.

28.1 Execution

28.1.1 db:stepInstruction()

Single step once at instruction level.

The result will only tell if the target was notified to start. An error result will indicate that the target is not believed to be in such a state that it can start.

To monitor progress and result of execution, you need to listen to notifications.

28.2 Breakpoint

28.2.1 db:breakDelete(number(s))

Remove one or more breakpoints.

The argument gives the number of the breakpoint(s) to remove. It can be either an integer or an array or integers.

28.2.2 db:breakInsert(location, &callBack)

Insert a new breakpoint.

The db table in the first argument may contain attributes to the breakpoint:

Table 28.1: Breakpoint attributes

field	type	description
temporary	boolean	set a temporary (one-time) breakpoint
hardware	boolean	this should 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 given thread

The second argument describes the location of the breakpoint, it can be:

Table 28.2: 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 first argument in the callBack is the object table that was used when the breakpoint was created. The second argument is just the breakpoint number. If the return value is a boolean false, then the breakpoint is not taken and execution resumes. This is done silently, there is no feedback about that we made a temporary stop to any user interface.

The call back will take place from the execution thread and it will block execution while it is running, so if you plan on continuing it is recommended that you return as quickly as you can.

If you are stopping, you can either do processing in the call back, or detect that we stop from the flow of notification. Keep in mind that if you do processing here, the user will not see any feedback that we stopped until you return.

Index