3 GCC-Based Software Development Tools

CONTENTS

This chapter presents the main components of a software development toolchain exclusively based on the GNU compiler collection (GCC) and other open-source projects. All of them are currently in widespread use, especially for embedded software development, but their documentation is sometimes highly technical and not easy to understand, especially for novice users.

After introducing the main toolchain components, the chapter goes into more detail on how they work and how they are used. This information has therefore the twofold goal of providing a short reference about the toolchain, and also of giving interested readers a sound starting point to help them move to the reference toolchain documentation in a smoother way. More detailed information about the compiler, mainly focusing on software portability aspects, will be given in Chapter 9.

The very last part of the chapter also gives an overview of how individual toolchain components are configured and built for a specific target architecture. A practical example of toolchain construction will be given in Chapter 12.

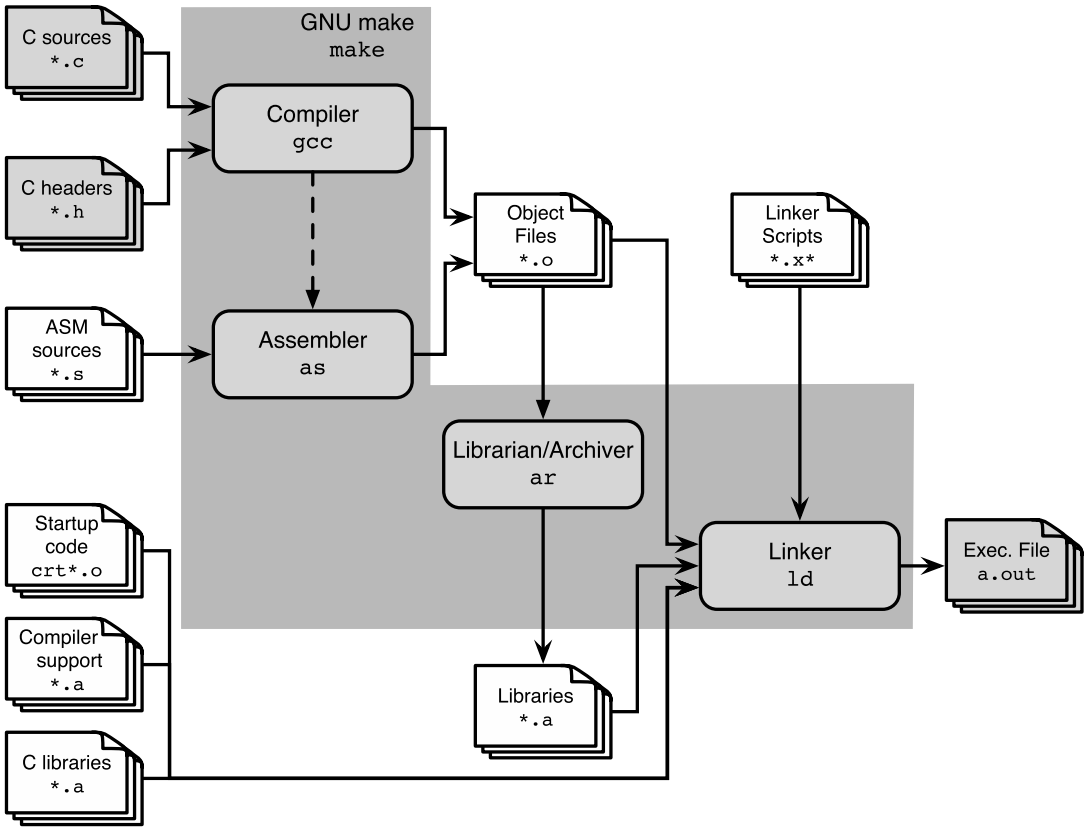

Figure 3.1 Overview of the toolchain workflow.

3.1 OVERVIEW

Generally speaking a toolchain is a complex set of software components. Collectively, they translate source code into executable machine code. Figure 3.1 outlines the general, theoretical toolchain workflow. As shown in the figure, its main components are:

1. The GCC *compiler* translates a C source file (that may include other headers and source files) and produces an object module. The translation process may require several intermediate code generation steps, involving the assembly language. In the last case, the compiler implicitly invokes the assembler `as` [55], too.
   The `gcc` program is actually a *compiler driver*. It is a programmable component, able to perform different actions by appropriately invoking other toolchain components, depending on the input file type (usually derived from its filename extension). The actions to be performed by `gcc`, as well as other options, are configured by means of a *specs* string or file. Both the compiler driver itself and its specs string are discussed in more detail in Section 3.2.
2. The `ar` *librarian* collects multiple object modules into a library. It is worth recalling that, especially in the past, the librarian was often called *archiver*, and the name of the toolchain component still derives from this term. The same tool also

performs many other operations on a library. For instance, it is able to extract or delete a module from it.
Other tools, like `nm` and `objdump`, perform more specialized operations related to object module contents and symbols. All these modules will not be discussed further in the following, due to space limitations. Interested readers may refer to their documentation [138] for more information.

3. The *linker* LD presented in Section 3.4, links object modules together and against libraries guided by one or more *linker scripts*. It resolves cross references to eventually build an *executable image*. Especially in small-scale embedded systems, the linking phase usually brings "user" and "system" code together into the executable image.
4. There are several categories of system code used at link time:
   - The *startup* object files `crt*.o` contain code that is executed first, when the executable image starts up. In standalone executable images, they also include system and hardware initialization code, often called *bootstrap* code.
   - The *compiler support library* `libgcc.a` contains utility functions needed by the code generator but too big/complex to be instantiated inline. For instance, integer multiply/divide or floating-point operations on processors without hardware support for them.
   - *Standard C libraries*, `libc.a` and `libm.a`, to be briefly discussed in Section 3.5.
   - Possibly, the *operating system* itself. This is the case of most small-scale operating systems. FREERTOS, the real-time operating system to be discussed and taken as a reference in Chapter 5, belongs to this category.
5. Last, but not least, another open-source component, *GNU make*, is responsible for coordinating the embedded software build process as a whole and automating it. In Figure 3.1 it is shown as a dark gray background that encompasses the other toolchain components and it will be the subject of Section 3.7.

An interesting twist of open-source toolchains is that the toolchain components are themselves written in a high-level language and are distributed as *source code*. For instance, the GNU compiler for the C programming language GCC is itself written in C. Therefore, a working C compiler is required in order to build GCC. There are several different ways to solve this "chicken and egg" problem, often called *bootstrap problem*, depending on the kind of compiler and build to be performed.

However, in embedded software development, the kind of toolchain most frequently used is the *cross-compilation* toolchain. As outlined in Chapter 2, cross compilation is the process of generating code for a certain architecture (the *target*) by means of a special toolchain, which runs on a different architecture (the development system, or *host*). This is because, due to limitations concerning their memory capacity and processor speed, embedded systems often lack the capability of compiling their own code.

The *bootstrap problem* is somewhat simpler to solve in this case, because it is possible to use a native toolchain on the host to build the cross-compilation toolchain.

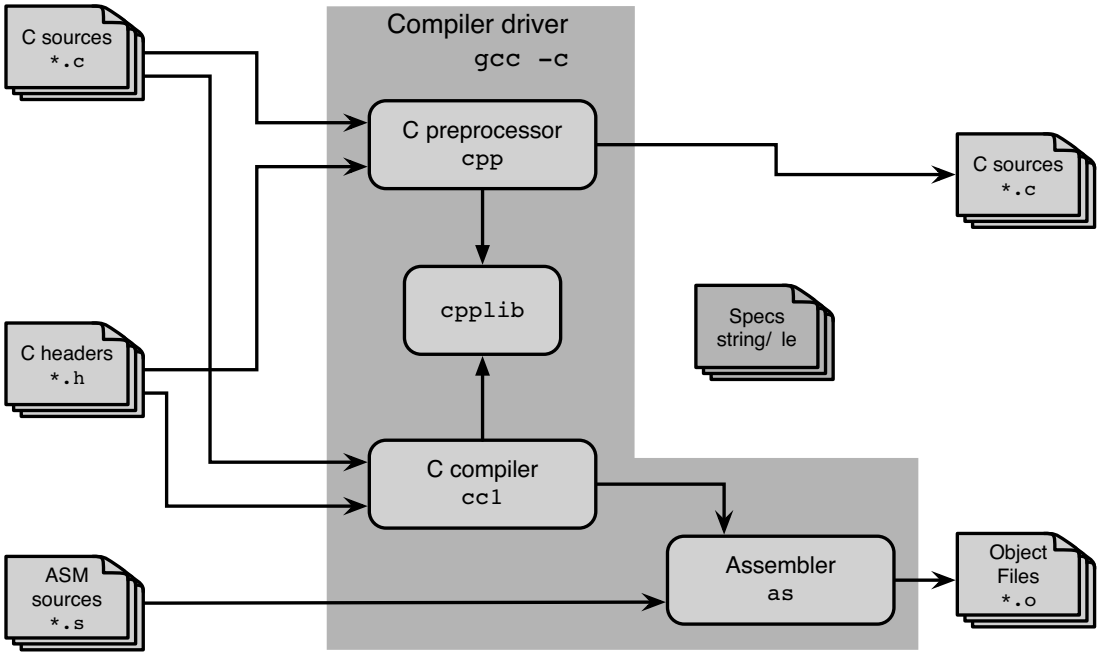

Figure 3.2 Overview of the compiler driver workflow.

The availability of a native toolchain is usually not an issue because most open-source operating system distributions already provide one ready-to-use.

3.2 COMPILER DRIVER WORKFLOW

The general outline of the compiler driver workflow is shown in Figure 3.2. According to the abstract view given in the previous section, when the compiler driver is used to compile a C source file, in theory it should perform the following steps:

1. preprocess the source code,
2. compile the source code into assembly code, and
3. assemble it to produce the output object file.

These steps are implemented by separate components, that is:

1. the C preprocessor `cpp`, better presented in Section 3.3,
2. the C compiler proper `cc1`, and
3. the assembler `as`.

The internal structure of the compiler deserves a chapter by itself due to its complexity. It will be more deeply discussed in Chapter 11, with particular emphasis on how it is possible to tune its workflow, in order to improve the performance and footprint of the code it generates.

In the following, we will focus instead on some peculiarities of the GCC-based toolchain that deviate from the abstract view just presented, as well as on the compiler driver itself.

First of all, as shown in the figure, the preprocessor is *integrated* in the C compiler and both are implemented by `cc1`. A *standalone preprocessor* `cpp` does exist, but it is not used during normal compilation.

In any case, the behavior of the standalone preprocessor and the one implemented in the compiler is consistent because both make use of the same preprocessing library, `cpplib`, which can also be used directly by application programs as a general-purpose macro expansion tool.

On the contrary, the assembler is implemented as a separate program, `as`, that is not part of the GCC distribution. Instead, it is distributed as part of the binary utilities package BINUTILS [138]. It will not be further discussed here due to space constraints.

One aspect peculiar to the GCC-based toolchain is that the compiler driver is programmable. Namely, it is driven by a set of *rules*, contained in a "specs" string or file. The specs string can be used to customize the behavior of the compiler driver. It ensures that the compiler driver is as flexible as possible, within its design envelope.

In the following, due to space constraints, we will just provide an overview of the expressive power that specs strings have, and illustrate what can be accomplished with their help, by means of a couple of examples. A thorough documentation of specs string syntax and usage can be found in [159].

First of all, the rules contained in a specs string specify which sequence of programs the compiler driver should run, and their arguments, depending on the kind of file provided as input. A default specs string is built in the compiler driver itself and is used when no custom specs string is provided elsewhere.

The sequence of steps to be taken in order to compile a file can be specified depending on the *suffix* of the file itself.

Other rules, associated with some command-line options may change the arguments passed by the driver to the programs it invokes.

```
*link: %{mbig-endian:-EB}
```

For example, the specs string fragment listed above specifies that if the command-line option `-mbig-endian` is given to the compiler driver, then the linker must be invoked with the `-EB` option.

Let us now consider a different specs string fragment:

```
*startfile: crti%O%s crtbegin%O%s new_crt0%O%s
```

In this case, the specs string specifies which object files should be unconditionally included at the start of the link. The list of object files is held in the `startfile` variable, mentioned in the left-hand part of the string, while the list itself is in the right-hand part, after the colon (`:`). It is often useful to modify the default set of objects in order to add language-dependent or operating system-dependent files without forcing programmers to mention them explicitly whenever they link an executable image.

More specifically:

- The `*startfile:` specification *overrides* the internal specs variable `startfile` and gives it a new value.

- `crti%O%s` and `crtbegin%O%s` are the standard initialization object files typical of C language programs. Within these strings,
  - `%O` represents the default suffix of object files. By default it is expanded to `.o` on Linux-based development hosts.
  - `%s` specifies that the object file is a system file and shall be located in the system search path rather than in user-defined directories.
- `new_crt0%O%s` replaces one of the standard initialization files of the C compiler to provide, as said previously, operating-system, language or machine-specific initialization functions.

3.3 C PREPROCESSOR WORKFLOW

The preprocessor, called `cpp` in a GNU-based toolchain, performs three main activities that, at least conceptually, take place *before* the source code is passed to the compiler proper. They are:

1. File inclusion, invoked by the `#include` directive.
2. Macro definition, by means of the `#define` directive, and expansion.
3. Conditional inclusion/exclusion of part of the input file from the compilation process, depending on whether or not some macros are defined (for instance, when the `#ifdef` directive is used) and their value (`#if` directive).

As it is defined in the language specification, the preprocessor works by *plaintext substitution*. However, Since the preprocessor and the compiler grammars are the same at the *lexical (token) level*, as an optimization, in a GCC-based toolchain the preprocessor also performs tokenization on the input files. Hence, it provides tokens instead of plaintext to the compiler.

A token [1] is a data structure that contains its *text* (a sequence of characters), some information about its *nature* (for instance whether the token represents a number, a keyword, or an identifier) and *debugging information* (file and line number it comes from).

Therefore, a token conveys additional information with respect to the portion of plaintext it corresponds to. This information is needed by the compiler anyway to further process its input, and hence, passing tokens instead of plaintext avoids duplicated processing.

Figure 3.3 contains a simplified view of the preprocessor workflow. Informally speaking, as it divides the input file into tokens, the preprocessor checks all of them and carries out one of three possible actions.

1. When the input token is a *preprocessor keyword*, like `#define` or `#include`, the preprocessor analyzes the tokens that follow it to build a complete statement and then obeys it.
   For example, after `#define`, it looks for a macro name, followed by the (optional) macro body. When the macro definition statement is complete, the preprocessor records the association *name* $\rightarrow$ *body* in a *table* for future use.

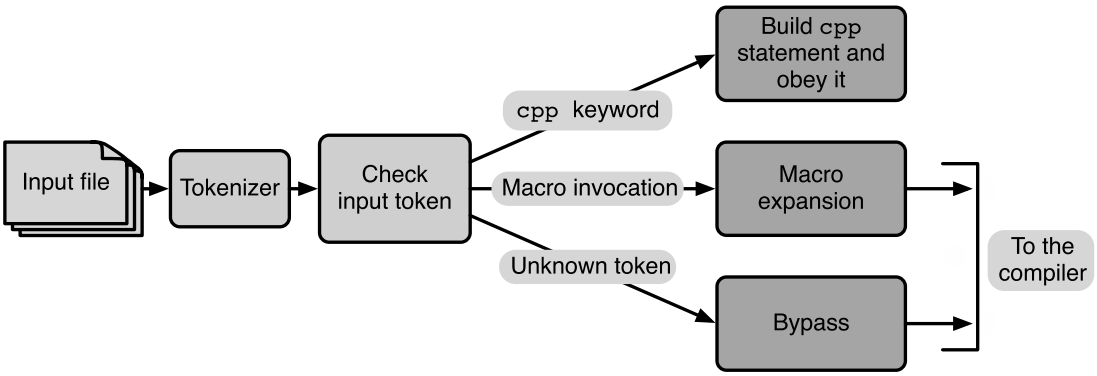

Figure 3.3 Simplified C preprocessor workflow.

It is useful to remark that, in this case, neither the preprocessor keyword, nor the following tokens (macro name and body in this example) are forwarded to the compiler. The macro *body* will become visible to the compiler only if the macro will be expanded later.
The *name* $\rightarrow$ *body* table is initialized when preprocessing starts and *discarded* when it ends. As a consequence, macro definitions are not kept across multiple compilation units. Initially, the table is not empty because the preprocessor provides a number of predefined macros.

2. When a macro is invoked, the preprocessor performs *macro expansion*. In the simplest case—that is, for *object-like* macros—macro expansion is triggered by encountering a macro name in the source code.
The macro is expanded by replacing its name with its body. Then, the result of the expansion is examined again to check whether or not further macro expansions can be done. When no further macro expansions can be done, the sequence of tokens obtained by the preprocessor as a result is forwarded to the compiler *instead of* the tokens that triggered macro expansion.
3. Tokens unknown to the preprocessor are simply passed to the compiler without modification. Since the preprocessor's and compiler's grammars are very different at the *syntax level*, many kinds of token known to the compiler have no meaning to the preprocessor, even though the latter is perfectly able to build the token itself.
For instance, it is obvious that type definitions are extremely important to the compiler, but they are completely transparent to the preprocessor.

The syntax of preprocessor keywords is fairly simple. In fact, they always start with a *sharp character* (`#`) in column one. Spaces are allowed between `#` and the rest of the keyword. The main categories of keyword are:

- Macro definition: `#define`.
- File inclusion: `#include`.
- Conditional compilation: `#ifdef`, `#ifndef`, `#if`, `#else`, `#elif`, and `#endif`.
- Other, for instance: `#warning` and `#error`.

Interested readers may refer to the full preprocessor documentation—that comes together with the GCC compiler documentation [160]—for further information about the other categories. In the following, we will mainly focus on the macro definition and expansion process.

There are two kinds of macros: *object-like* and *function-like* macros. Object-like macros are the simplest and their handling by the preprocessor can be summarized in two main points.

- The name of the macro is replaced by its body when it is encountered in the source file. The result is reexamined after expansion to check whether or not other macros are involved. If this is the case, they are expanded as well.
- Macro expansion does *not* take place when a macro is defined, but only when the macro is used. Hence, it is possible to have *forward references* to other macros within macro definitions.

For example, it is possible to define two macros, `A` and `B`, as follows.

```
#define B A+3
#define A 12
```

Namely, the definition of macro `B` does not produce any error although `A` has not been defined yet. When `B` is encountered in the source file, after the previously listed definitions, it is expanded as: `B` → `A+3` → `12+3`.

Since no other macro names are present in the intermediate result, macro expansion ends at this point and the three tokens `12`, `+`, and `3` are forwarded to the compiler.

Due to the way of communication between the preprocessor and the compiler, explained previously and outlined in Figure 3.3, the compiler does not know how tokens are obtained. For instance, it cannot distinguish between tokens coming from macro expansion and tokens taken directly from the source file.

As a consequence, if `B` is used within a more complex expression, the compiler might get confused and interpret the expression in a counter-intuitive way. Continuing the previous example, the expression `B*5` is expanded by the preprocessor as `B*5` → `A+3*5` → `12+3*5`.

When the compiler parses the result, the evaluation of `3*5` is performed *before* `+`, due to the well-known precedence rules of arithmetic operators, although this may not be the behavior the programmer expects.

To solve this problem, it is often useful to put additional parentheses around macro bodies, as is shown in the following fragment of code.

```
#define B (A+3)
#define A 12
```

In this way, when `B` is invoked, it is expanded as: `B` → `(A+3)` → `(12+3)`. Hence, the expression `B*5` is seen by the compiler as `(12+3)*5`. In other words, the additional pair of parentheses coming from the expansion of macro `B` establishes the

boundaries of macro expansion and overrides the arithmetic operator precedence rules.

The second kind of macros is represented by function-like macros. The main differences with respect to object-like macros just discussed can be summarized as follows.

- Function-like macros have a list of *parameters*, enclosed between parentheses `()`, after the macro name in their definition.
- Accordingly, they must be invoked by using the macro name followed by a list of *arguments*, also enclosed between `()`.

To illustrate how function-like macro expansion takes place, let us consider the following fragment of code as an example.

```
#define F(x, y) x*y*K
#define K 7
#define Z 3
```

When a function-like macro is invoked, its arguments are completely macro-expanded *first*. Therefore, for instance, the first step in the expansion of `F(Z, 6)` is: `F(Z, 6)` → `F(3, 6)`.

Then, the parameters in the macro body are replaced by the corresponding, expanded arguments. Continuing our example:

- parameter `x` is replaced by argument `3`, and
- `y` is replaced by `6`.

After the replacement, the body of the macro becomes `3*6*K`. At this point, the modified body replaces the function-like macro invocation. Therefore, `F(3, 6)` → `3*6*K`.

The final step in function-like macro expansion consists of re-examining the result and check whether or not other macros (either object-like or function-like macros) can be expanded. In our example, the result of macro expansion obtained so far still contains the object-like macro name `K` and the preprocessor expands it according to its definition: `3*6*K` → `3*6*7`.

To summarize, the complete process of macro expansion when the function-like macro `F(Z, 6)` is invoked is

```
F(Z, 6) → F(3, 6)                    (argument expansion)
→ 3*6*K                              (parameter substitution in the macro body)
→ 3*6*7                              (expansion of the result)
```

As already remarked previously about object-like macros, parentheses may be useful around parameters in function-like macro bodies, too, for the same reason.

For instance, the expansion of `F(Z, 6+9)` proceeds as shown below and clearly produces a counter-intuitive result, if we would like to consider `F` to be akin to a mathematical function.

```
F(Z, 6+9) → F(3, 6+9)                    (argument expansion)
→ 3*6+9*K                                (parameter substitution in the macro body)
→ 3*6+9*7                                (expansion of the result)
```

It is possible to work around this problem by defining `F(x, y)` as `(x)*(y)*(K)`. In this way, the final result of the expansion is:

$$\texttt{F(Z, 6+9)} \rightarrow \cdots \rightarrow \texttt{(3)*(6+9)*(7)}$$

as intended.

3.4 THE LINKER

The main purpose of the linker, usually called LD in the GNU toolchain and fully described in [34], is to *combine* a number of object files and libraries and place them into an *executable image*. In order to do this, the linker carries out two main activities:

1. it *resolves* inter-module symbol references, and
2. it *relocates* code and data.

The GNU linker may be invoked directly as `<cross>-ld`, where `<cross>` is the prefix denoting the target architecture when cross-compiling. However, as described in Section 3.2, it is more often called by the compiler driver `<cross>-gcc` automatically as required. In both cases, the linking process is driven by a *linker script* written in the Link Editor Command Language.

Before describing in a more detailed way how the linker works, let us briefly recall a couple of general linker options that are often useful, especially for embedded software development.

- The option `-Map=<file>` writes a *link map* to `<file>`. When the additional `--cref` option is given, the map also includes a cross reference table. Even though no further information about it will be given here, due to lack of space, the link map contains a significant amount of information about the outcome of the linking process. Interested readers may refer to the linker documentation [34] for more details about it.
- The option `--oformat=<format>` sets the format of the output file, among those recognized by `ld`. Being able to precisely control the output format helps to upload the executable image into the target platform successfully. Reference [34] contains the full list of supported output formats, depending on the target architecture and linker configuration.
- The options `--strip` and `--strip-debug` remove symbolic information from the output file, leaving only the executable code and data. This step is sometimes required for executable image upload tools to work correctly, because they might not handle any extra information present in the image properly.

Symbolic information is mainly used to keep debugging information, like the mapping between source code line numbers and machine code. For this reason, it is anyway unnecessary to load them into the target memory.

When `ld` is invoked through the compiler driver, linker options must be preceded by the escape sequence `-Wl` to distinguish them from options directed to the compiler driver itself. A comma is used to separate the escape sequence from the string to be forwarded to the linker and no intervening spaces are allowed. For instance, `gcc -Wl,-Map=f.map -o f f.c` compiles and links `f.c`, and gives the `-Map=f.map` option to the linker.

As a last introductory step, it is also important to informally recall the main differences between *object modules*, *libraries*, and *executable images* as far as the linker is concerned. These differences, outlined below, will be further explained and highlighted in the following sections.

- Object files are *always* included in the final executable image, instead the object modules found in libraries (and also called library modules) are used only *on demand*.
- More specifically, library modules are included by the linker only if they are needed to resolve pending symbol references, as will be better described in the following section.
- A library is simply a collection of unmodified object modules put together into a single file by the archiver or librarian `ar`.
- An executable image is formed by binding together object modules, either standalone or from libraries, by the linker. However, it is not simply a collection, like a library is, because the linker performs a significant amount of work in the process.

3.4.1 SYMBOL RESOLUTION AND RELOCATION

Most linker activities revolve around *symbol* manipulation. Informally speaking, a symbol is a convenient way to refer to the address of an object in memory in an *abstract* way (by means of a human-readable name instead of a number) and even before its exact *location* in memory is known.

The use of symbols is especially useful to the compiler during code generation. For example, when the compiler generates code for a backward jump at the end of a loop, two cases are possible:

1. If the processor supports *relative* jumps—that is, a jump in which the target address is calculated as the sum of the current program counter plus an offset stored in the jump instruction—the compiler may be able to generate the code completely and automatically by itself, because it knows the "distance" between the jump instruction and its target. The linker is not involved in this case.
2. If the processor only supports *absolute* jumps—that is, a jump in which the target address is directly specified in the jump instruction—the compiler must leave a

"blank" in the generated code, because it does not know where the code will eventually end up in memory. As will be better explained in the following, this blank will be filled by the linker when it performs symbol resolution and relocation.

Another intuitive example, regarding *data* instead of *code* addresses, is represented by global variables accessed by means of an `extern` declaration. Also in this case, the compiler needs to refer to the variable *by name* when it generates code, without knowing its memory address at all. Also in this case, the code that the compiler generates will be incomplete because it will include "blanks," in which symbols are referenced instead of actual memory addresses.

When the linker collects object files, in order to produce the executable image, it becomes possible to associate symbol *definitions* and the corresponding *references*, by means of a name-matching process known as *symbol resolution* or (according to an older nomenclature) *snapping*.

On the other hand, symbol values (to continue our examples, addresses of variables, and the exact address of machine instructions) become known when the linker *relocates* object contents in order to lay them out into memory. At this point, the linker can "fill the blanks" left by the compiler.

As an example of how symbol resolution takes place for data, let us consider the following two, extremely simple source files.

f.c

```
extern int i;

void f(void) {
    i = 7;
}
```

g.c

```
int i;
```

In this case, symbol resolution proceeds as follows.

- When the compiler generates code for `f()` it does not know where (and if) variable `i` is defined. Therefore, in `f.o` the address of `i` is left blank, to be filled by the linker.
- This is because the compiler works on exactly *one* source file at a time. When it is compiling `f.c` it does not consider `g.c` in any way, even though both files appear together on the command line.
- During symbol resolution, the linker observes that `i` is defined in `g.o` and associates the definition with the reference made in `f.o`.
- After the linker relocates the contents of `g.o`, the address of `i` becomes known and can eventually be used to complete the code in `f.o`.

It is also useful to remark that initialized data need a special treatment when the initial values must be in non-volatile memory. In this case, the linker must cooperate with the startup code (by providing memory layout information) so that the initial-

ization can be performed correctly. Further information on this point will be given in Section 3.4.3.

3.4.2 INPUT AND OUTPUT SEQUENCES

As mentioned before, the linking process is driven by a set of commands, specified in a linker script. A linker script can be divided into three main parts, to be described in the following sections. Together, these three parts fully determine the overall behavior of the linker, because:

1. The *input and output* part picks the input files (object files and libraries) that the linker must consider and directs the linker output where desired.
2. The *memory layout* part describes the position and size of all memory banks available on the target system, that is, the space the linker can use to lay out the executable image.
3. The *section and memory mapping* part specifies how the input files contents must be mapped and relocated into memory banks.

If necessary, the linker script can be split into multiple files that are then bound together by means of the `INCLUDE <filename>` directive. The directive takes a file name as argument and directs the linker to include that file "as if" its contents appeared in place of the directive itself. The linker supports nested inclusion, and hence, `INCLUDE` directives can appear both in the main linker script and in an included script.

This is especially useful when the linker script becomes complex or it is convenient to divide it into parts for other reasons, for instance, to distinguish between architecture or language-dependent parts and general parts.

Input and output linker script commands specify:

- Which *input* files the linker will operate on, either object files or libraries. This is done by means of one or more `INPUT()` commands, which take the names of the files to be considered as arguments.
- The *sequence* in which they will be scanned by the linker, to perform symbol resolution and relocation. The sequence is implicitly established by the order in which input commands appear in the script.
- The special way a specific file or group of files will be handled. For instance, the `STARTUP()` command labels a file as being a startup file rather than a normal object file.
- Where to look for *libraries*, when just the library name is given. This is accomplished by specifying one or more search paths by means of the `SEARCH_DIR()` command.
- Where the *output*—namely, the file that contains the executable image—goes, through the `OUTPUT()` command.

Most of these commands have a command-line counterpart that, sometimes, is more commonly used. For instance, the `-o` command-line option acts the same as

`OUTPUT()` and mentioning an object file name on the linker command line has the same effect as putting it in an `INPUT()` linker script command.

The *entry point* of the executable image—that is, the instruction that shall be executed first—can be set by means of the `ENTRY(<symbol>)` command in the linker script, where `<symbol>` is a symbol.

However, it is important to remark that the *only* effect of `ENTRY` is to *keep a record* of the desired entry point and store it into the executable image itself. Then, it becomes the responsibility of the *bootloader* mentioned in Chapter 2—often simply called *loader* in linker's terminology—to obey what has been requested in the executable image.

When no loader is used—that is, the executable image is uploaded by means of an upload tool residing on the development host, and then runs on the target's "bare metal"—the entry point is defined by *hardware*. For example, most processors start execution from a location indicated by their *reset vector* upon powerup. Any entry point set in the executable image is ignored in this case.

All together, the *input* linker script commands eventually determine the linker *input sequence*. Let us now focus on a short fragment of a linker sequence that contains several input commands and describe how the input sequence is built from them.

```
INPUT(a.o, b.o, c.o)
INPUT(d.o, e.o)
INPUT(libf.a)
```

Normally, the linker scans the input files *once* and in the order established by the input sequence, which is defined by:

- The order in which files appear *within* the `INPUT()` command. In this case, `b.o` follows `a.o` in the input sequence and `e.o` follows `d.o`.
- If there are multiple `INPUT()` commands in the linker script, they are considered in the same sequence as they appear in the script.

Therefore, in our example the linker scans the files in the order: `a.o`, `b.o`, `c.o`, `d.o`, `e.o`, and `libf.a`.

As mentioned previously, object files can also be specified on the linker command line and become part of the input sequence, too. In this case:

- The command line may also include an option (`-T`) to refer to the linker script.
- The input files specified on the command line are combined with those mentioned in the linker script depending on *where* the linker script has been referenced.

For instance, if the command line is `gcc ... a.o -Tscript b.o` and the linker script `script` contains the command `INPUT(c.o, d.o)`, then the input sequence is: `a.o`, `c.o`, `d.o`, and `b.o`.

As mentioned previously the *startup file* is a special object file because it contains low-level hardware initialization code. For example, it may set the CPU clock source

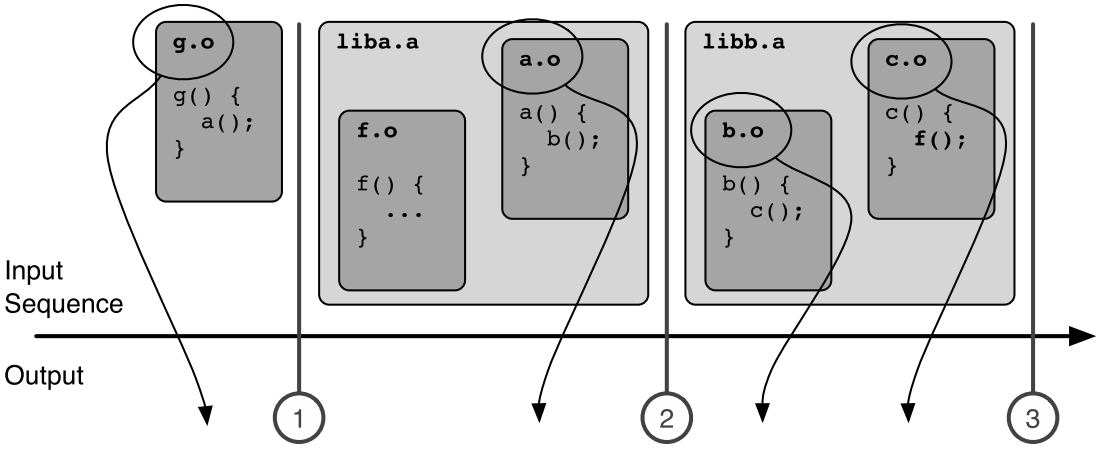

Figure 3.4 Linker's handling of object files and libraries in the input sequence.

and frequency. Moreover, it sets up the execution environment for application code. For instance, it is responsible of preparing initialized data for use. As a consequence, its position in memory may be constrained by the hardware startup procedure.

The `STARTUP(<file>)` command forces `<file>` to be the very first object file in the input sequence, regardless of where the command is. For example, the linker script fragment

```
INPUT(a.o, b.o)
STARTUP(s.o)
```

leads to the input sequence `s.o`, `a.o`, and `b.o`, even though `s.o` is mentioned last.

Let us now mention how the linker transforms the input sequence into the *output sequence* of object modules that will eventually be used to build the executable image. We will do this by means of an example, with the help of Figure 3.4. In our example, the input sequence is composed of an object file `g.o` followed by two libraries, `liba.a` and `libb.a`, in this order. They are listed at the top of the figure, from left to right. For clarity, libraries are depicted as lighter gray rectangles, while object files correspond to darker gray rectangles. In turn, object files contain function definitions and references, as is also shown in the figure.

The construction of the output sequence proceeds as follows.

- Object module `g.o` is unconditionally placed in the output sequence. In the figure, this action is represented as a downward-pointing arrow. As a consequence the symbol `a`, which is referenced from the body of function `g()`, becomes undefined at point ①.
- When the linker scans `liba.a`, it finds a definition for `a` in module `a.o` and resolves it by placing `a.o` into the output. This makes `b` undefined at point ②, because the body of `a` contains a reference to it.

- Since only `a` is undefined at the moment, only module `a.o` is put in the output. More specifically, module `f.o` is not, because the linker is not aware of any undefined symbols related to it.
- When the linker scans `libb.a`, it finds a definition of `b` and places module `b.o` in the output. In turn, `c` becomes undefined. Since `c` is defined in `c.o`, that is, another module within the *same library*, the linker places this object module in the output, too.
- Module `c.o` contains a reference to `f`, and hence, `f` becomes undefined. Since the linker scans the input sequence only once, it is unable to refer back to `liba.a` at this point. Even though `liba.a` defines `f`, that definition is not considered. At point ③ `f` is still undefined.

According to the example it is evident that the linker implicitly handles libraries as *sets*. Namely, the linker picks up object modules from a set *on demand* and places them into the output. If this action introduces additional undefined symbols, the linker looks into the set *again*, until no more references can be resolved. At this time, the linker moves to the next object file or library.

As also shown in the example, this default way of scanning the input sequence is problematic when libraries contain *circular cross references*. More specifically, we say that a certain library *A* contains a circular cross-reference to library *B* when one of *A*'s object modules contains a reference to one of *B*'s modules and, symmetrically, one of *B*'s modules contains a reference back to one module of library *A*.

When this occurs, regardless of the order in which libraries *A* and *B* appear in the input sequence, it is always possible that the linker is unable to resolve a reference to a symbol, even though one of the libraries indeed contains a definition for it. This is what happens in the example for symbol `f`.

In order to solve the problem, it is possible to *group* libraries together. This is done by means of the command `GROUP()`, which takes a list of libraries as argument. For example, the command `GROUP(liba.a, libb.a)` groups together libraries `liba.a` and `libb.a` and instructs the linker to handle both of them as a *single set*.

Going back to the example, the effect of `GROUP(liba.a, libb.a)` is that it directs the linker to look back into the set, find the definition of `f`, and place module `f.o` in the output.

It is possible to mix `GROUP()` and `INPUT()` within the input sequence to transform just *part* of it into a set. For example, given the input sequence listed below:

```
INPUT(a.o, b.o)
GROUP(liba.a, libb.a)
INPUT(libc.a)
```

the linker will first examine `a.o`, and then `b.o`. Afterwards, it will handle `liba.a` and `libb.a` as a single set. Last, it will handle `libc.a` on its own.

Moreover, as will become clearer in the following, the use of `GROUP()` makes sense only for libraries, because object files are handled in a different way in the first place. Figure 3.5 further illustrates the differences. In particular, the input sequence

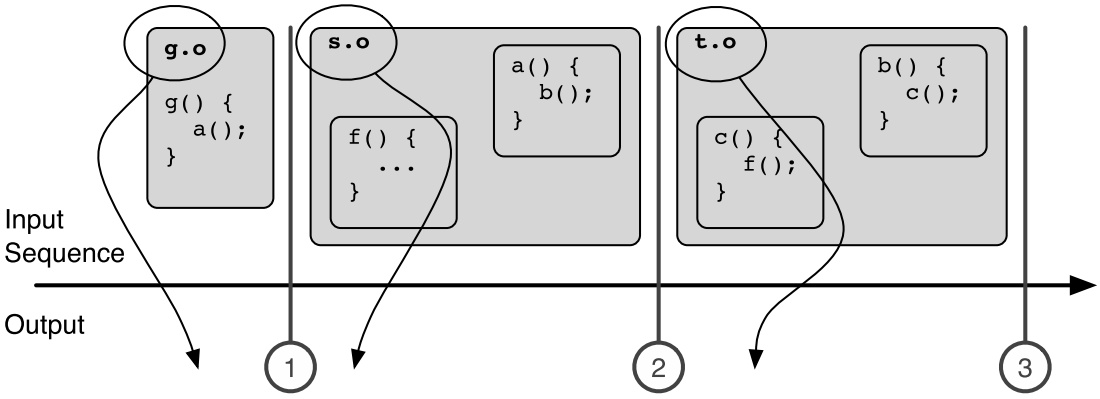

Figure 3.5 Object files versus libraries.

shown in Figure 3.5 is identical to the one previously considered in Figure 3.4, with the only exception that libraries have been replaced by object modules.

The input sequence of Figure 3.5 is processed as follows:

- Object module `g.o` is placed in the output and symbol `a` becomes undefined at point ①.
- When the linker scans `s.o` it finds a definition for `a` and places *the whole object module* in the output.
- This provides a definition of `f` even though it was not called for at the moment and makes `b` undefined at point ②.
- When the linker scans `t.o` it finds a definition of `b` and it places the whole module in the output. This also provides a definition of `c`.
- The reference to `f` made by `c` can be resolved successfully because the output already contains a definition of `f`.

As a result, there are no unresolved symbols at point ③. In other words, circular references between object files are resolved automatically because the linker places them into the output as a whole.

3.4.3 MEMORY LAYOUT

The `MEMORY { ... }` command is used to describe the memory layout of the target system as a set of *blocks* or *banks*. For clarity, command contents are usually written using one line of text for each block. Its general syntax is:

```
MEMORY
{
    <name> [(<attr>)] : ORIGIN = <origin>, LENGTH = <len>
    ...
}
```

where:

- `ORIGIN` and `LENGTH` are keywords of the linker script.
- `<name>` is the name given to the block, so that the other parts of the linker script can refer to that memory block by name.
- `<attr>` is optional. It gives information about the type of memory block and affects which kind of information the linker is allowed to allocate into it. For instance, `R` means read-only, `W` means read/write, and `X` means that the block may contain executable code.
- `<origin>` is the starting address of the memory block.
- `<len>` is the length, in *bytes*, of the memory block.

For example, the following `MEMORY` command describes the Flash memory bank and the main RAM bank of the LPC1768, as defined in its user manual [128].

```
MEMORY
{
    rom (rx)  : ORIGIN = 0x00000000, LENGTH = 512K
    ram (rwx) : ORIGIN = 0x10000000, LENGTH =  32K
}
```

To further examine a rather common issue that may come even from the seemingly simple topic of memory layout, let us now consider a simple definition of an initialized, global variable in the C language, and draw some comments on it. For example,

```
int a = 3;
```

- After the linker allocates variable `a`, it resides somewhere in RAM memory, for instance, at address `0x1000`. RAM memory is needed because it must be possible to modify the value of `a` during program execution.
- Since RAM memory contents are not preserved when the system is powered off, its initial value `3` must be stored in nonvolatile memory instead, usually within a Flash memory bank. Hence, the initial value is stored, for instance, at address `0x0020`.
- In order to initialize `a`, the value `3` must be copied from Flash to RAM memory. In a standalone system, the copy must be performed by either the bootloader or the startup code, but in any case *before* the main C program starts. This is obviously necessary because the code generated by the compiler *assumes* that initialized variables indeed contain their initial value. The whole process is summarized in Figure 3.6.
- To setup the initialized global variable correctly, the linker associates *two* memory addresses to initialized global data. Address `0x0020` is the *Load Memory Address* (LMA) of `a`, because this is the memory address where its *initial contents* are stored.
- The second address, `0x1000` in our example, is the *virtual memory address* (VMA) of `a` because this is the address used by the processor to refer to `a` at *runtime*.

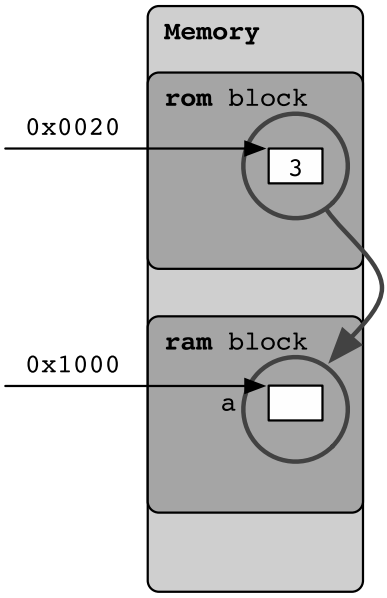

Figure 3.6 Initialized variables setup, VMA and LMA.

Often, the VMA and LMA of an object are *the same*. For example, the address where a function is stored in memory is the same address used by the CPU to call it. When they are not, a copy is necessary, as illustrated previously.

This kind of copy can *sometimes* be avoided by using the `const` keyword of the C language, so that read-only data are allocated only in ROM. However, this is not strictly guaranteed by the language specification, because `const` only determines the data *property* at the language level, but does not necessarily affect their *allocation* at the memory layout level.

In other words, data properties express how they can be manipulated in the program, which is not directly related to where they are in memory. As a consequence, the relationship between these two concepts may or may not be kept by the toolchain during object code generation.

From the practical point of view, it is important to remark that the linker follows the *same order* when it allocates memory for initialized variables in RAM and when it stores their initial value in ROM. Moreover, the linker does not interleave any additional memory object in either case. As a consequence, the layout of the ROM area that stores initial values and of the corresponding RAM area is the same. Only their starting addresses are different.

In turn, this implies that the *relative position* of variables and their corresponding initialization values within their areas is the same. Hence, instead of copying variable by variable, the startup code just copies the whole area in one single sweep.

The base addresses and size of the RAM and ROM areas used for initialized variables are provided to the startup code, by means of symbols defined in the linker script as described in Section 3.4.4.

As a final note for this section, it is worth remarking that there is an unfortunate clash of terminology between virtual memory addresses as they are defined in the

linker's nomenclature and virtual memory addresses in the context of virtual memory systems, outlined in Chapter 15.

3.4.4 LINKER SCRIPT SYMBOLS

As described previously, the concept of *symbol* plays a central role in linker's operations. Symbols are mainly *defined* and *referenced* by object files, but they can also be defined and referenced in a *linker script*. There is just one "category" of symbols. Namely:

- A symbol defined in an object module can be referenced by the linker script. In this way, the object module can modify the inner workings of the linker script itself and affect section mapping and memory layout, just by defining symbols appropriately. For example, it is possible to set the *stack size* of the executable image from one of the object modules.
- Symmetrically, a symbol defined in the linker script can be referenced by an object module, and hence, the linker script can determine some aspects of the object module's behavior. For example, as mentioned in Section 3.4.3, the linker script can communicate to the startup code the base addresses and size of the RAM and ROM areas used for initialized variables.

An *assignment*, denoted by means of the usual = (equal sign) operator, gives a value to a symbol. The value is calculated as the result of an expression written on the right-hand side of the assignment. The expression may contain most C-language arithmetic and Boolean operators. It may involve both constants and symbols.

The result of an expression may be *absolute* or *relative* to the beginning of an output section, depending on the expression itself (mainly, the use of the `ABSOLUTE()` function) and also *where* the expression is in the linker script.

The special (and widely used) symbol `.` (dot) is the *location counter*. It represents the absolute or relative output location (depending on the context) that the linker is about to fill while it is scanning the linker script—and its input sequence in particular—to lay out objects into memory. With some exceptions, the location counter may generally appear whenever a normal symbol is allowed. For example, it appears on the right-hand side of an assignment in the following example.

```
__stack = .
```

This assignment sets the symbol `__stack` to the value of the location counter. Assigning a value to `.` moves the location counter. For example, the following assignment:

```
. += 0x4000
```

allocates `0x4000` bytes starting from where the location counter currently points and moves the location counter itself after the reserved area.

An assignment may appear in three different positions in a linker script and its position partly affects how the linker interprets it.

1. *By itself.* In this case, the assigned value is absolute and, contrary to the general rule outlined previously, the location counter `.` cannot be used.
2. As a statement *within a* `SECTIONS` *command*. The assigned value is *absolute* but, unlike in the previous case, the use of `.` is allowed. It represents an *absolute* location counter.
3. Within an *output section description*, nested in a `SECTIONS` command. The assigned value is *relative* and `.` represents the *relative* value of the location counter with respect to the beginning of the output section.

As an example, let us consider the following linker script fragment. More thorough and formal information about output sections is given in Section 3.4.5.

```
SECTIONS
{
  .  = ALIGN(0x4000);
  . += 0x4000;
  __stack = .;
}
```

In this example:

- The first assignment aligns the location counter to a multiple of 16Kbyte (`0x4000`).
- The second assignment moves the location counter forward by 16Kbyte. That is, it allocates 16Kbyte of memory for the stack.
- The third assignment sets the symbol `__stack` to the top of the stack. The startup code will refer to this symbol to set the initial stack pointer.

3.4.5 SECTION AND MEMORY MAPPING

The contents of each input object file are divided by the *compiler* (or assembler) into several categories according to their characteristics, like:

- code (`.text`),
- initialized data (`.data`),
- uninitialized data (`.bss`).

Each category corresponds to its own *input section* of the object file, whose name has also been listed above. For example, the object code generated by the C compiler is placed in the `.text` section of the input object files. Libraries follow the same rules because they are just collections of object files.

The part of linker script devoted to *section mapping* tells the linker how to fill the memory image with *output sections*, which are generated by collecting *input sections*. It has the following syntax:

```
SECTIONS
{
    <sub-command>
    ...
}
```

where:

- The `SECTIONS` command encloses a sequence of *sub-commands*, delimited by braces.
- A sub-command may be:
  - an `ENTRY` command, used to set the initial entry point of the executable image as described in Section 3.4.2,
  - a symbol assignment,
  - an overlay specification (seldom used in modern programs),
  - a *section mapping* command.

A section mapping command has a relatively complex syntax, illustrated in the following.

```
<section> [<address>] [(<type>)] :
  [<attribute> ...]
  [<constraint>]
  {
    <output-section-command>
    ...
  }
  [> <region>] [AT> <lma_region>]
  [: <phdr> ...] [= <fillexp>]
```

Most components of a section mapping command, namely, the ones shown within brackets (`[]`), are optional. Within a section mapping command, an *output section command* may be:

- a *symbol assignment*, outlined in Section 3.4.4,
- *data values* to be included directly in the output section, mainly used for padding,
- a special *output section keyword*, which will not be further discussed in this book,
- an *input section* description, which identifies which input sections will become part of the output section.

An input section description must be written according to the syntax indicated below.

```
<filename> ( <section_name> ... )
```

It indicates which input sections must be mapped into the output section. It consists of:

- A `<filename>` specification that identifies one or more object files in the input sequence. Some wildcards are allowed, the most common one is `*`, which matches *all* files in the input sequence. It is also possible to *exclude* some input files, by means of `EXCLUDE_FILE(...)`, where `...` is the list of files to be excluded. This is useful in combination with wildcards, to refine the result produced by the wildcards themselves.

- One or more <section_name> specifications that identify which input sections, within the files indicated by <filename>, we want to refer to.

The order in which input section descriptions appear is important because it sets the order in which input sections are placed in the output sections.

For example, the following input section description:

```
* ( .text .rodata )
```

places the .text and .rodata sections of all files in the input sequence in the output section. The sections appear in the output in the same order as they appear in the input.

Instead, these slightly different descriptions:

```
* ( .text )
* ( .rodata )
```

first places all the .text sections, and then all the .rodata sections.

Let us now examine the other main components of the section mapping command one by one. The very first part of a section mapping command specifies the output section *name*, *address*, and *type*. In particular:

- <section> is the name of the output section and is mandatory.
- <address>, if specified, sets the *VMA* of the output section. When it is not specified, the linker sets it *automatically*, based on the output memory block <region>, if specified, or the current location counter. Moreover, it takes into account the strictest alignment constraint required by the input sections that are placed in the output sections and the output sections alignment itself, which will be specified with an [<attribute>] and will be explained later.
- The most commonly used special output section <type> is NOLOAD. It indicates that the section shall not be loaded into memory when the program is run. When omitted, the linker creates a normal output section specified with the section name, for instance, .text.

Immediately thereafter, it is possible to specify a set of output section *attributes*, according to the following syntax:

```
[AT( <lma> )]
[ALIGN( <section_align> )]
[SUBALIGN( <subsection_align> )]
[<constraint>]
```

- The AT attribute sets the *LMA* of the output section to address <lma>.
- The ALIGN attribute specifies the alignment of the output section.
- The SUBALIGN attribute specifies the alignment of the input sections placed in the output section. It overrides the “natural” alignment specified in the input sections themselves.

- `<constraint>` is normally empty. It may specify under which constraints the output sections must be created. For example, it is possible to specify that the output section must be created only if all input sections are read-only [34].

The *memory block mapping* specification is the very last part of a section mapping command and comes after the list of output section commands. It specifies in which memory block (also called *region*) the output section must be placed. Its syntax is:

```
[> <region>] [AT> <lma_region>]
[: <phdr> ...] [= <fillexp>]
```

where:

- `> <region>` specifies the memory block for the output section *VMA*, that is, where it will be referred to by the processor.
- `AT> <lma_region>` specifies the memory block for the output section *LMA*, that is, where its contents are loaded.
- `<phdr>` and `<fillexp>` are used to assign the output section to an *output segment* and to set the *fill pattern* to be used in the output section, respectively.

Segments are a concept introduced by some executable image formats, for example the executable and linkable format (ELF) [33] format. In a nutshell, they can be seen as groups of sections that are considered as a single unit and handled all together by the loader.

The fill pattern is used to fill the parts of the output section whose contents are not explicitly specified by the linker script. This happens, for instance, when the location counter is moved or the linker introduces a gap in the section to satisfy an alignment constraint.

3.5 THE C RUNTIME LIBRARY

Besides defining the language itself, the international standards concerning the C programming language [87, 88, 89, 90] also specify a number of *library functions*. They range from very simple ones, like `memcpy` (that copies the contents of one memory area of known size into another) to very complex ones, like `printf` (that converts and print out its arguments according to a rich format specification).

The implementation of these functions is not part of the compiler itself and is carried out by another toolchain component, known as *C runtime library*. Library functions are themselves written in C and distributed as source code, which is then compiled into a library by the compiler during toolchain generation. Another example is the *C math library* that implements all floating-point functions specified by the standards except basic arithmetic.

Among the several C runtime libraries available for use for embedded software development, we will focus on NEWLIB [145], an open-source component widely used for this purpose.

The most important aspect that should be taken into account when porting this library to a new architecture or hardware platform, is probably to configure and provide an adequate *multitasking support* to it. Clearly, this is of utmost importance to ensure that the library operates correctly when it is used in a multitasking environment, like the one provided by any real-time operating system.

In principle, NEWLIB is a *reentrant* C library. That is, it supports multiple tasks making use of and calling it *concurrently*. However, in order to do this, two main constraints must be satisfied:

1. The library must be able to maintain some *per-task information*. This information is used, for instance, to hold the per-task `errno` variable, I/O buffers, and so on.
2. When multiple tasks share information through the library, it is necessary to implement critical regions within the library itself, by means of appropriate synchronization points. This feature is used, for instance, for dynamic memory allocation (like the `malloc` and `free` functions) and I/O functions (like `open`, `read`, `write`, and others).

It is well known that, in most cases, when a C library function fails, it just returns a Boolean error indication to the caller or, in other cases, a `NULL` pointer. Additional information about the reason for the failure is stored in the `errno` variable.

In a *single-task* environment, there may only be up to one pending library call at any given time. In this case, `errno` can be implemented as an ordinary global variable. On the contrary, in a *multitasking* environment, multiple tasks may make a library call concurrently. If `errno` were still implemented as a global variable, it would be impossible to know which task the error information is for. In addition, an error caused by a task would overwrite the error information used by all the others.

The issue cannot be easily solved because, as will be better explained in Chapters 4 and 5, in a multitasking environment the task execution order is nondeterministic and may change from time to time.

As a consequence, the `errno` variable must hold per-task information and must not be shared among tasks. In other words, one instance of the `errno` variable must exist for each thread, but it must still be "globally accessible" in the same way as a global variable. The last aspect is important mainly for backward compatibility. In fact, many code modules were written with single-thread execution in mind and preserving backward compatibility with them is important.

By "globally accessible," we mean that a given thread must be allowed to refer to `errno` from *anywhere* in its code, and the expected result must be provided.

The method adopted by NEWLIB to address this problem, on single-core systems, is depicted in Figure 3.7 and is based on the concept of *impure pointer*.

Namely, a *global* pointer (`impure_ptr`) allows the library to refer to the *per-task data structure* (`struct _reent`) of the running task. In order to do this, the library relies on appropriate support functions (to be provided in the *operating system support layer*), which:

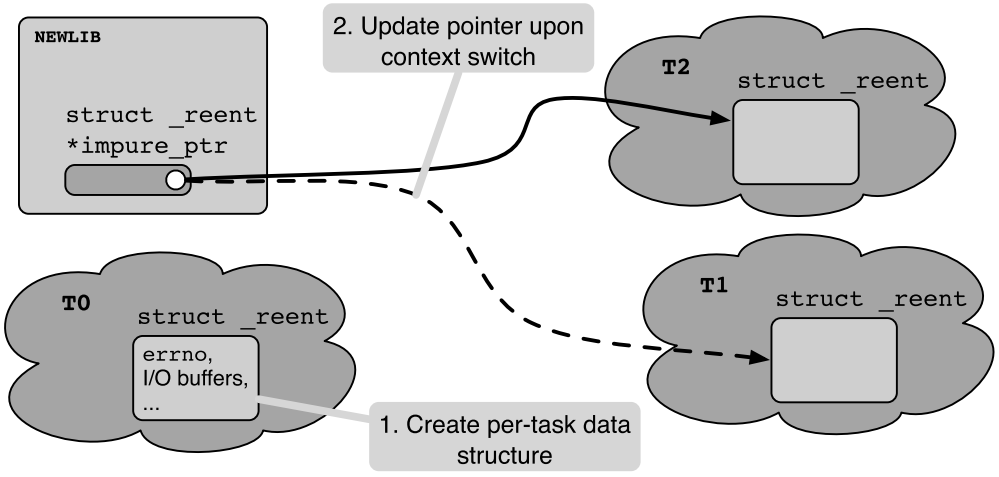

Figure 3.7 Handling of the NEWLIB impure pointer on single-core systems.

1. Create and destroy per-task data structures at an appropriate time, that is, task creation and deletion. This is possible because—even though the internal structure of the `struct _reent` is opaque to the operating system—its size is known.
2. Update `impure_ptr` upon every context switch, so that it always points to the per-task data structure corresponding to the running task.

In addition, some library modules need to maintain data structures that are shared among all tasks. For instance, the dynamic memory allocator maintains a single memory pool that is used by the library itself (on behalf of tasks) and is also made available to tasks for direct use by means of `malloc()`, `free()`, and other functions.

The formal details on how to manage concurrent access to shared resources, data structures in this case, will be given in Chapter 5. For the time being, we can consider that, by intuition, it is appropriate to let only one task at a time use those data structures, to avoid corrupting them. This is accomplished by means of appropriate *synchronization points* that force tasks to wait until the data structures can be accessed safely.

As shown in the left part of Figure 3.8, in this case, the synchronization point is *internal* to the library. The library implements synchronization by calling an operating system support function at critical region boundaries. For instance, the dynamic memory allocation calls `__malloc_lock()` before working on the shared memory pool, and `__malloc_unlock()` after it is done.

Therefore, a correct synchronization is a shared responsibility between the library and the operating system, because:

- the library "knows" where critical regions are and what they are for, and
- the operating system "knows" how to synchronize tasks correctly.

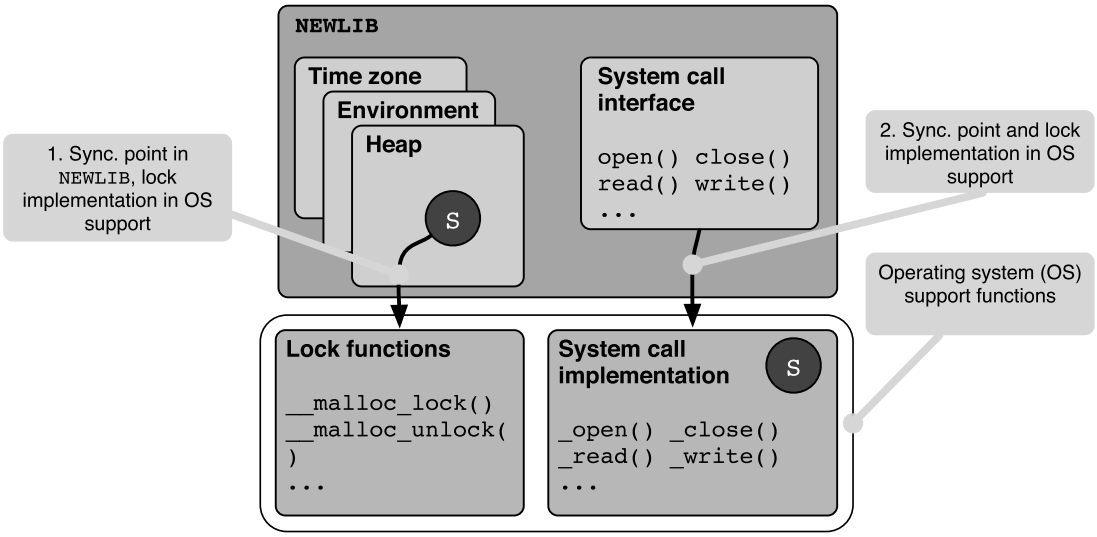

Figure 3.8 NEWLIB synchronization points.

The operating system support functions, to be implemented when the library is ported to a new architecture, have the crucial role of *gluing* these two parts together.

As shown in the right side of Figure 3.8, synchronization is also needed when tasks share other kinds of resource, like *files* and *devices*. For those resources, which historically were not managed by the library itself, resource management (and therefore synchronization) is completely *delegated* to other components, like the operating system itself or another specialized library that implements a filesystem.

Accordingly, for functions like `read()` and `write()`, the library only encapsulates the implementation provided by other components without adding any additional feature. Since these functions are often implemented as system calls, NEWLIB refers to them by this name in the documentation, even though this may or may not be true in an embedded system.

For instance, in Chapter 12 we will show an example of how to implement a USB-based filesystem by means of an open-source filesystem library and other open-source components. In that case, as expected and outlined here, the filesystem library requires support for synchronization to work correctly when used in a multitasking context.

3.6 CONFIGURING AND BUILDING OPEN-SOURCE SOFTWARE

It is now time to consider how individual toolchain components are built from their source code. In order to obtain a working toolchain component, three main phases are needed:

1. The *configuration* phase sets up all the parameters that will be used in the build phase and automatically generates a `Makefile`. As will be explained in Section 3.7, this file contains key information to automate the build process.

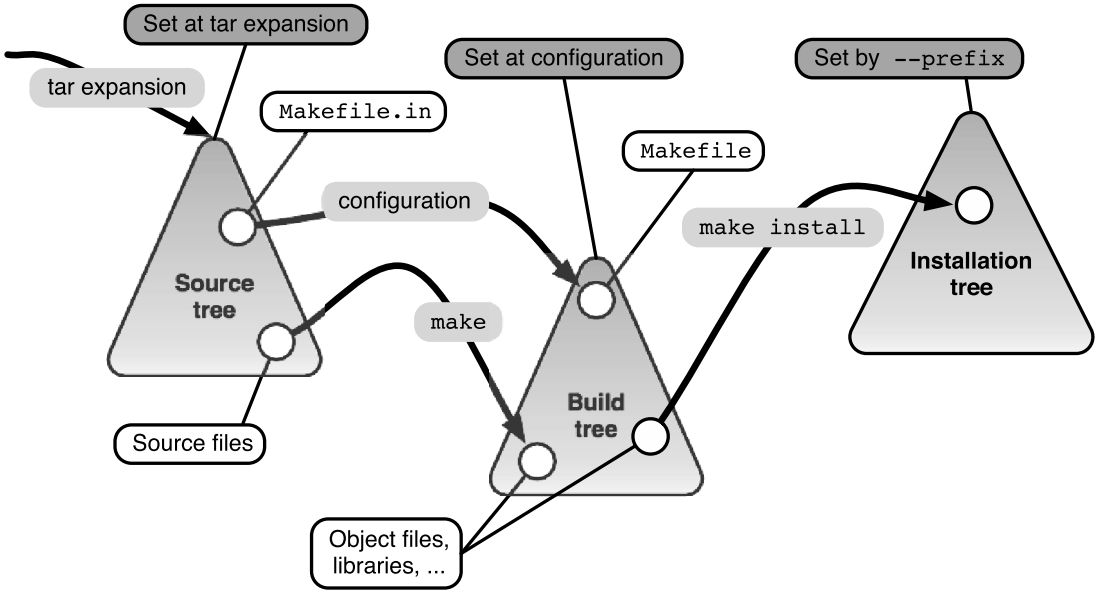

Figure 3.9 Summary of the toolchain components installation process.

Examples of parameters provided during configuration are the architecture that the toolchain component must target, where the component shall be installed on the development host, which extra libraries shall be used, and so on.

2. During the *build* phase, orchestrated by GNU make, intermediate files and byproducts as well as the final results are generated from source files.
3. After a successful build, the results are moved to their final location during the *installation* phase, started manually. In this way, it is possible to rebuild the toolchain multiple times without overwriting installed versions of the same component.

As shown in Figure 3.9, it is crucial to distinguish among three different kinds of directories that will be used during the build of each toolchain component:

- The *source* directory tree is the place where the source code of the component is stored, when its distribution archive is unpacked.
- The *build* directory tree is used in the build phase. The build phase is started by running GNU make. As a result, the corresponding `make` command must be launched from there. In turn, `make` uses the `Makefile` generated during the configuration phase. In this case, the configuration must also be run in the same place.
- The *installation* directory tree is used to install the toolchain component. It can be specified during the configuration phase, by means of the `--prefix` argument. Installation is started explicitly by means of the `make install` command.

The first preliminary step required to build a toolchain component is to create the source tree in the current directory, starting from the source package of the component, as shown in the following.

```
tar xf <source_package>
```

Then, we can create the build directory, change into it and configure the component, specifying where the installation tree is, as follows.

```
mkdir <build_dir>
cd <build_dir>
<path_to_source_tree>/configure \
    --prefix=<installation_tree> \
    ...<other_options>...
```

Finally, we can build and (if the build was successful) install the component.

```
make
make install
```

Each toolchain component must know its own installation prefix, that is, the root of the directory tree into which they will be installed. As shown previously, this information must be provided to the component at configuration time.

Although it is possible, in principle, to install each toolchain component in a different location, it is recommended to use a single installation prefix for all of them. A convenient way to do it is to set the environment variable `PREFIX` to a suitable path, and use it when configuring the toolchain components.

Moreover, most toolchain components need to locate and run executable modules belonging to other components, not only when they are used normally, but also while they are being built. For example, the build process of GCC also builds the C runtime library. This requires the use of the assembler, which is part of BINUTILS.

As toolchain components are built, their executable files are stored into the `bin` subdirectory under the installation prefix. The value of environment variable `PATH` must be extended to include this new path. This is because, during the build process, executable files are located (as usual) by consulting the `PATH` environment variable.

In the following, we will give more detailed information on how to build a working toolchain component by component. It is assumed that both the source code packages have already been downloaded into the current working directory. Moreover, if any toolchain component requires architecture or operating system-dependent patches, it is assumed that they are applied according to the instructions provided with them.

The first component to be built is BINUTILS, which contains the GNU binary utilities, by means of the following sequence of commands.

```
tar xjf binutils-<version>.tar.bz2

mkdir binutils-<version>-b

cd binutils-<version>-b

../binutils-<version>/configure --target=<target> \
    --prefix=$PREFIX --enable-multilib --disable-nls

make
```

In the previous listing:

- The most important configuration parameter is the intended target architecture, indicated by `<target>`.
- Similarly, `<version>` indicates the version of the BINUTILS package we intend to build.
- The `--prefix` argument specifies where BINUTILS will be installed.
- `--enable-multilib` activates support for multiple architecture and variant-dependent libraries.
- Finally, `--disable-nls` disables the national language support to make the build process somewhat faster.

The `--disable-werror` option allows the build process to continue after a warning. It may be useful when the host toolchain is much more recent than the toolchain being built, and hence, it produces unforeseen warnings. However, it should be used with care because it implicitly applies to the whole build. Therefore, it may mask real issues with the build itself.

The next two components to be configured and built are GCC and NEWLIB, by means of the following sequence of commands.

```
tar xjf gcc-<version>.tar.bz2
tar xzf newlib-<version>.tar.gz

cd gcc-<version>
tar xzf ../gmp-<version>.tar.gz
mv gmp-<version> gmp
tar xjf ../mpfr-<version>.tar.bz2
mv mpfr-<version> mpfr
ln -s ../newlib-<version>/newlib .
cd ..

mkdir gcc-<version>-b
cd gcc-<version>-b
../gcc-<version>/configure \
    --target=<target> --prefix=$PREFIX \
    --with-gnu-as --with-gnu-ld \
    --with-newlib \
    --enable-languages="c,c++" \
    --enable-multilib \
    --disable-shared --disable-nls

make
make install
```

In this sequence of commands:

- Like before, the GCC and NEWLIB source packages are expanded into two separate directories.

- In order to build GCC, two extra libraries are required: `gmp` (for multiple-precision integer arithmetic) and `mpfr` (for multiple-precision floating-point arithmetic). They are used at *compile-time*—that is, the executable image is *not* linked against them—to perform all arithmetic operations involving *constant* values present in the source code. Those operations must be carried out in extended precision to avoid round-off errors. The required version of these libraries is specified in the GCC installation documentation, in the `INSTALL/index.html` file within the GCC source package. The documentation also contains information about where the libraries can be downloaded from.
- The GCC configuration script is also able to configure `gmp` and `mpfr` automatically. Hence, the resulting `Makefile` builds all three components together and statically links the compiler with the two libraries. This is done if their source distributions are found in two subdirectories of the GCC source tree called `gmp` and `mpfr`, respectively, as shown in our example.
- Similarly, both GCC and NEWLIB can be built in a single step, too. To do this, the source tree of NEWLIB must be linked to the GCC source tree. Then, configuration and build proceed in the usual way. The `--with-newlib` option informs the GCC configuration script about the presence of NEWLIB.
- The GCC package supports various programming languages. The `--enable-languages` option can be used to restrict the build to a subset of them, only C and C++ in our example.
- The `--disable-shared` option disables the generation of *shared libraries*. Shared libraries are nowadays very popular in general-purpose operating systems, as they allow multiple applications to share the same memory-resident copy of a library and save memory. However, they are not yet commonly used in embedded systems, especially small ones, due to the significant amount of runtime support they require to dynamically link applications against shared libraries when they are launched.

3.7 BUILD PROCESS MANAGEMENT: GNU MAKE

The GNU make tool [65], fully described in [157], manages the *build process* of a software component, that is, the execution of the correct sequence of commands to transform its source code modules into a library or an executable program as efficiently as possible.

Since, especially for large components, it rapidly becomes unfeasible to rebuild the whole component every time, GNU make implements an extensive inference system that allows it to:

1. decide which parts of a component shall be rebuilt after some source modules have been updated, based on their *dependencies*, and then
2. automatically execute the appropriate sequence of *commands* to carry out the rebuild.

Both dependencies and command sequences are specified by means of a set of *rules*, according to the syntax that will be better described in the following. These rules can be defined either *explicitly* in a GNU make input file (often called `Makefile` by convention), and/or *implicitly*. In fact, `make` contains a rather extensive set of predefined, built-in rules, which are implicitly applied unless overridden in a `Makefile`.

GNU make retrieves the explicit, user-defined rules from different sources. The main ones are:

- One of the files `GNUmakefile`, `makefile`, or `Makefile` if they are present in the current directory. The first file found takes precedence on the others, which are silently neglected.
- The file specified by means of the `-f` or `--file` options on the command line.

It is possible to include a `Makefile` into another by means of the `include` directive. In order to locate the file to be included, GNU make looks in the current directory and any other directories mentioned on the command line, using the `-I` option. As will be better explained in the following, the `include` directive accepts any file name as argument, and even names computed on the fly by GNU make itself. Hence, it allows programmers to use any arbitrary file as (part of) a `Makefile`.

Besides options, the command line may also contain additional *arguments*, which specify the *targets* that GNU make must try to update. If no targets are given on the command line, GNU make pursues the *first* target explicitly mentioned in the `Makefile`.

It is important to remark that the word *target*, used in the context of GNU make, has a different meaning than the same word in the context of open-source software configuration and build, described in Section 3.6. There, target is the target architecture for which a software component must be built, whereas here it represent a goal that GNU make is asked to pursue.

3.7.1 EXPLICIT RULES

The general format of an *explicit* rule in a `Makefile` is:

```
<target> ... : <prerequisites> ...
        <command line>
        ...
```

In an explicit rule:

- The `target` is usually a file that will be (re)generated when the rule is applied.
- The `prerequisites` are the files on which `target` depends and that, when modified, trigger the regeneration of the target.
- The sequence of `command lines` are the actions that GNU make must perform in order to regenerate the target, in shell syntax.

Table 3.1
GNU make Command Line Execution Options

Option	Description
`@`	Suppress the automatic *echo* of the command line that GNU make normally performs immediately before execution.
`-`	When this option is present, GNU make ignores any *error* that occurs during the execution of the command line and continues.

- Every command line must be preceded by a *tab* and is executed in *its own* shell.

It is extremely important to pay attention to the last aspect of command line execution, which is often neglected, because it may have very important consequences on the effects commands have.

For instance, the following rule does *not* list the contents of directory `somewhere`.

```
all:
        cd somewhere
        ls
```

This is because, even though the `cd` command indeed changes current directory to `somewhere` within the shell it is executed by, the `ls` command execution takes place in a new shell, and the previous notion of current directory is lost when the new shell is created.

As mentioned previously, the prerequisites list specifies the *dependencies* of the target. GNU make looks at the prerequisites list to deduce *whether or not* a target must be regenerated by applying the rule. More specifically, GNU make applies the rule when one or more prerequisites are *more recent* than the target. For example, the rule:

```
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
```

specifies that the object file `kbd.o` (target) must be regenerated when at least one file among `kbd.c defs.h command.h` (prerequisites) has been modified. In order to regenerate `kbd.o`, GNU make invokes `cc -c kbd.c` (command line) within a shell.

The shell, that is, the command line interpreter used for command line execution is by default `/bin/sh` on unix-like systems, unless the `Makefile` specifies otherwise by setting the `SHELL` variable. Namely, it does not depend on the user login shell to make it easier to port the `Makefile` from one user environment to another.

Unless otherwise specified, by means of one of the command line execution options listed in Table 3.1, commands are *echoed* before execution. Moreover, when an *error* occurs in a command line, GNU make abandons the execution of the current rule and (depending on other command-line options) may stop completely. Command line execution options must appear at the very beginning of the command line, before the text of the command to be executed. In order to do the same things in a systematic way GNU make supports options like `--silent` and `--ignore`, which apply to all command lines or, in other words, change the default behavior of GNU make.

3.7.2 VARIABLES

A variable is a name defined in a `Makefile`, which represents a text string. The string is the *value* of the variable. The value of a certain variable `VAR`—by convention, GNU make variables are often written in all capitals—is usually retrieved and used (that is, *expanded*) by means of the construct `$(VAR)` or `${VAR}`.

In a `Makefile`, variables are expanded "on the fly," while the file is being read, except when they appear within a command line or on the right-hand part of variable assignments made by means of the assignment operator "=". The last aspect of variable expansion is important and we will further elaborate on it in the following, because the behavior of GNU make departs significantly from what is done by most other language processors, for instance, the C compiler.

In order to introduce a dollar character somewhere in a `Makefile` without calling for variable expansion, it is possible to use the escape sequence `$$`, which represents *one* dollar character, `$`.

Another difference with respect to other programming languages is that the `$()` operators can be nested. For instance, it is legal, and often useful, to state `$($(VAR))`. In this way, the *value* of a variable (like `VAR`) can be used as a variable *name*. For example, let us consider the following fragment of a `Makefile`.

```
MFLAGS = $(MFLAGS_$(ARCH))
MFLAGS_Linux  = -Wall -Wno-attributes -Wno-address
MFLAGS_Darwin = -Wall
```

- The variable `MFLAGS` is set to different values depending on the contents of the `ARCH` variable. In the example, this variable is assumed to be set elsewhere to the host operating system name, that is, either `Linux` or `Darwin`.
- In summary, this is a *compact* way to have different, operating system-dependent compiler flags in the variable `MFLAGS` without using conditional directives or write several separate `Makefile`s, one for each operating system.

A variable can get a value in several different, and rather complex ways, listed here in order of decreasing priority.

Table 3.2
GNU make Assignment Operators

Operator	Description
`VAR = ...`	Define a recursively-expanded variable
`VAR := ...`	Define a simply-expanded variable
`VAR ?= ...`	Defines the recursively-expanded variable `VAR` only if it has not been defined already
`VAR += ...`	Append `...` to variable `VAR` (see text)

1. As specified when GNU make is *invoked*, by means of an assignment statement put directly on its command line. For instance, the command `make VAR=v` invokes GNU make with `VAR` set to `v`.
2. By means of an *assignment* in a `Makefile`, as will be further explained in the following.
3. Through a shell *environment* variable definition.
4. Moreover, some variables are set *automatically* to useful values during rule application.
5. Finally, some variables have an *initial* value, too.

When there are multiple assignments to the same variable, the highest-priority one *silently* prevails over the others.

GNU make supports two kinds, or *flavors* of variables. It is important to comment on this difference because they are *defined* and *expanded* in different ways.

1. *Recursively-expanded* variables are defined by means of the operator =, informally mentioned previously. The evaluation of the right-hand side of the assignment, as well as the expansion of any references to other variables it may contain, are delayed until the variable being defined is itself expanded. The evaluation and variable expansion then proceed recursively.
2. *Simply-expanded* variables are defined by means of the operator `:=`. The value of the variable is determined once and for all when the assignment is executed. The expression on the right-hand side of the assignment is evaluated immediately, expanding any references to other variables.

Table 3.2 lists all the main assignment operators that GNU make supports. It is worth mentioning that the "append" variant of the assignment preserves (when possible) the kind of variable it operates upon. Therefore:

- If `VAR` is undefined it is the same as =, and hence, it defines a recursively expanded variable.

- If `VAR` is already defined as a simply expanded variable, it immediately expands the right-hand side of the assignment and appends the result to the previous definition.
- If `VAR` is already defined as a recursively expanded variable, it appends the right-hand side of the assignment to the previous definition without performing any expansion.

In order to better grasp the effect of delayed variable expansion, let us consider the following two examples.

```
X = 3
Y = $(X)
X = 8
```

In this first example, the value of `Y` is `8`, because the right-hand side of its assignment is expanded only when `Y` is *used*. Let us now consider a simply expanded variable.

```
X = 3
Y := $(X)
X = 8
```

In this case, the value of `Y` is `3` because the right-hand side of its assignment is expanded immediately, when the assignment is performed.

As can be seen from the previous examples, delayed expansion of recursively expanded variables has unusual, but often useful, side effects. Let us just briefly consider the two main benefits of delayed expansion:

- Forward variable references in assignments, even to variables that are still undefined, are not an issue.
- When a variable is eventually expanded, it makes use of the "latest" value of the variables it depends upon.

3.7.3 PATTERN RULES

Often, all files belonging to the same *group* or *category* (for example, object files) follow the same generation rules. In this case, rather than providing an explicit rule for each of them and lose generality in the `Makefile`, it is more appropriate and convenient to define a *pattern rule*.

As shown in the following short code example, a pattern rule applies to all files that match a certain *pattern*, which is specified within the rule *in place of the target*.

```
%.o :  %.c
    cc -c $<

kbd.o :  defs.h command.h
```

In particular:

Table 3.3
GNU make Automatic Variables

Var.	Description	Example value
`$@`	Target of the rule	`kbd.o`
`$<`	*First* prerequisite of the rule	`kbd.c`
`$^`	List of *all* prerequisites of the rule, delimited by blanks	`kbd.c defs.h command.h`
`$?`	List of prerequisites that are *more recent* than the target	`defs.h`
`$*`	Stem of the rule	`kbd`

- Informally speaking, in the pattern the character `%` represents any non-empty character string.
- The same character can be used in the prerequisites, too, to specify how they are related to the target.
- The command lines associated with a pattern rule can be customized, based on the specific target the rule is being applied for, by means of *automatic variables* like `$<`.
- It is possible to augment the prerequisites of a pattern rule by means of explicit rules without command lines, as shown at the end of the example.

More precisely, a target pattern is composed of three parts: a *prefix*, a `%` character, and a *suffix*. The prefix and/or suffix may be empty. A target name (which often is a file name) matches the pattern if it starts with the pattern prefix and ends with the pattern suffix. The non-empty sequence of characters between the prefix and the suffix is called the *stem*.

Since, as said previously, rule targets are often file names, directory specifications in a pattern are handled specially, to make it easier to write compact and general rules. In particular:

- If a target pattern does not contain any *slash*—which is the character that separates directory names in a file path specification—all directory names are removed from target file names before comparing them with the pattern.
- Upon a successful match, directory names are restored at the beginning of the stem. This operation is carried out before generating prerequisites.
- Prerequisites are generated by substituting the stem of the rule in the right-hand part of the rule, that is, the part that follows the colon (`:`).
- For example, file `src/p.o` satisfies the pattern rule `%.o : %.c`. In this case, the prefix is *empty*, the stem is `src/p` and the prerequisite is `src/p.c` because the `src/` directory is removed from the file name before comparing it with the pattern and then restored.

When a pattern rule is applied, GNU make automatically defines several *automatic variables*, which become available in the corresponding command lines. Table 3.3 contains a short list of these variables and describes their contents.

As an example, the rightmost column of the table also shows the value that automatic variables would get if the rules above were applied to regenerate `kbd.o`, mentioned before, because `defs.h` has been modified.

To continue the example, let us assume that the `Makefile` we are considering contains the following additional rule. The rule updates library `lib.a`, by means of the `ar` tool, whenever any of the object files it contains (`main.o kbd.o disk.o`) is updated.

```
lib.a :  main.o kbd.o disk.o
    ar rs $@ $?
```

After applying the previous rule, `kbd.o` becomes more recent than `lib.a`, because it has just been updated. In turn, this triggers the application of the second rule shown above. While the second rule is being applied, the automatic variable corresponding to the target of the rule (`$@`) is set to `lib.a` and the list of prerequisites more recent than the target (`$?`) is set to `kbd.o`.

To further illustrate the use of automatic variables, we can also remark that we could use `$^` instead of `$?` in order to completely rebuild the library rather than update it. This is because, as mentioned in Table 3.3, `$^` contains the list of *all* prerequisites of the rule.

It is also useful to remark that GNU make comes with a large set of predefined, *built-in* rules. Most of them are pattern rules, and hence, they generally apply to a wide range of targets and it is important to be aware of their existence. They can be printed by means of the command-line option `--print-data-base`, which can also be abbreviated as `-p`.

For instance, there is a built-in pattern rule to generate an object file given the corresponding C source file:

```
%.o: %.c
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

The variables cited in the command line have got a built-in definition as well, that is:

```
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
OUTPUT_OPTION = -o $@
CC = cc
```

As a consequence, by appropriately defininining some additional variables, for example `CFLAGS`, it is often possible to customize the behavior of a built-in rule instead of defining a new one.

When a rule in the `Makefile` overlaps with a built-in rule because it has the same target and prerequisites, *the former* takes precedence. This priority scheme has been designed to avoid undue interference of built-in rules the programmer may be unaware of, with any rule he/she explicitly put into his/her `Makefile`.

3.7.4 DIRECTIVES AND FUNCTIONS

GNU make provides a rather extensive set of *directives* and built-in *functions*. In general, directives control how the input information needed by GNU make is built, by taking it from different input files, and which parts of those input files are considered. Provided here is a glance at two commonly-used directives, namely:

- The `include <file>` directive instructs GNU make to temporarily *stop reading* the current `Makefile` at the point where the directive appears, read the additional `file`(s), and then continue.
  The `file` specification may contain a single file name or a list of names, separated by spaces. In addition, it may also contain variable and function expansions, as well as any wildcards understood and used by the shell to match file names.
- The `ifeq (<exp1>, <exp2>)` directive evaluates the two expressions `exp1` and `exp2`. If they are textually identical, then GNU make uses the `Makefile` section between `ifeq` and the next `else` directive; otherwise it uses the section between `else` and `endif`.
  In other words, this directive is similar to conditional statements in other programming languages. Directives `ifneq`, `ifdef`, and `ifndef` also exist, and do have the expected, intuitive meaning.

Concerning functions, the general syntax of a function call is

```
$(<function> <arguments>)
```

where

- `function` represents the function name and `arguments` is a list of one or more arguments. At least one blank space is required to separate the function name from the first argument. Arguments are separated by commas.
- By convention, variable names are written in all capitals, whereas function names are in lowercase, to help readers distinguish between the two.

Arguments may contain references to:

- *Variables*, for instance: `$(subst a,b,$(X))`. This statement calls the function `subst` with 3 arguments: `a`, `b`, and the result of the expansion of variable `X`.
- Other, nested *function calls*, for example: `$(subst a,b,$(subst c,d,$(X)))`. Here, the third argument of the outer `subst` is the result of the inner, nested `subst`.

As for directives, in the following we are about to informally discuss only a few GNU make functions that are commonly found in `Makefile`s. Interested readers should refer to the full documentation of GNU make, available online [65], for in-depth information.

- The function `$(subst from,to,text)` replaces `from` with `to` in `text`. Both `from` and `to` must be simple text strings. For example:

  `$(subst .c,.o,p.c q.c)` ⟶ `p.o q.o`

- The function `$(patsubst from,to,text)` is similar to `subst`, but it is more powerful because `from` and `to` are patterns instead of text strings. The meaning of the `%` character is the same as in pattern rules.

  `$(patsubst %.c,%.o,p.c q.c)` ⟶ `p.o q.o`

- The function `$(wildcard pattern ...)` returns a list of names of existing files that match one of the given patterns. For example, `$(wildcard *.c)` evaluates to the list of all C source files in the current directory. `wildcard` is commonly used to set a variable to a list of file names with common characteristics, like C source files. Then, it is possible to further work on the list with the help of other functions and use the results as targets, as shown in the following example.

```
SRC = $(wildcard *.c)
ELF = $(patsubst %.c,%.elf,$(SRC))

all: $(ELF)

%.elf: %.c
	$(CC) -o $@ $<
```

- The function `$(shell command)` executes a shell `command` and captures its output as return value. For example, when executed on a Linux system:

  `$(shell uname)` ⟶ `Linux`

  In this way, it is possible to set a variable to an operating system-dependent value and have GNU make do different things depending on the operating system it is running on, without providing separate `Makefile`s for all of them, which would be harder to maintain.

3.8 SUMMARY

This chapter provided an overview of the most peculiar aspects of a GCC-based cross-compilation toolchain, probably the most commonly used toolchain for embedded software development nowadays.

After starting with an overview of the workflow of the whole toolchain, which was the subject of Section 3.1, the discussion went on by focusing on specific components, like the compiler driver (presented in Section 3.2), the preprocessor (described in Section 3.3), and the runtime libraries (Section 3.5).

Due to its complexity, the discussion of the inner workings of the compiler has been postponed to Chapter 11, where it will be more thoroughly analyzed in the context of performance and memory footprint optimization.

Instead, this chapter went deeper into describing two very important, but often neglected, toolchain components, namely the linker (which was the subject of Section 3.4) and GNU make (discussed in Section 3.7).

Last, but not least, this chapter also provided some practical information on how to configure and build the toolchain components, in Section 3.6. More specific information on how to choose the exact version numbers of those components will be provided in Chapter 12, in the context of a full-fledged case study.