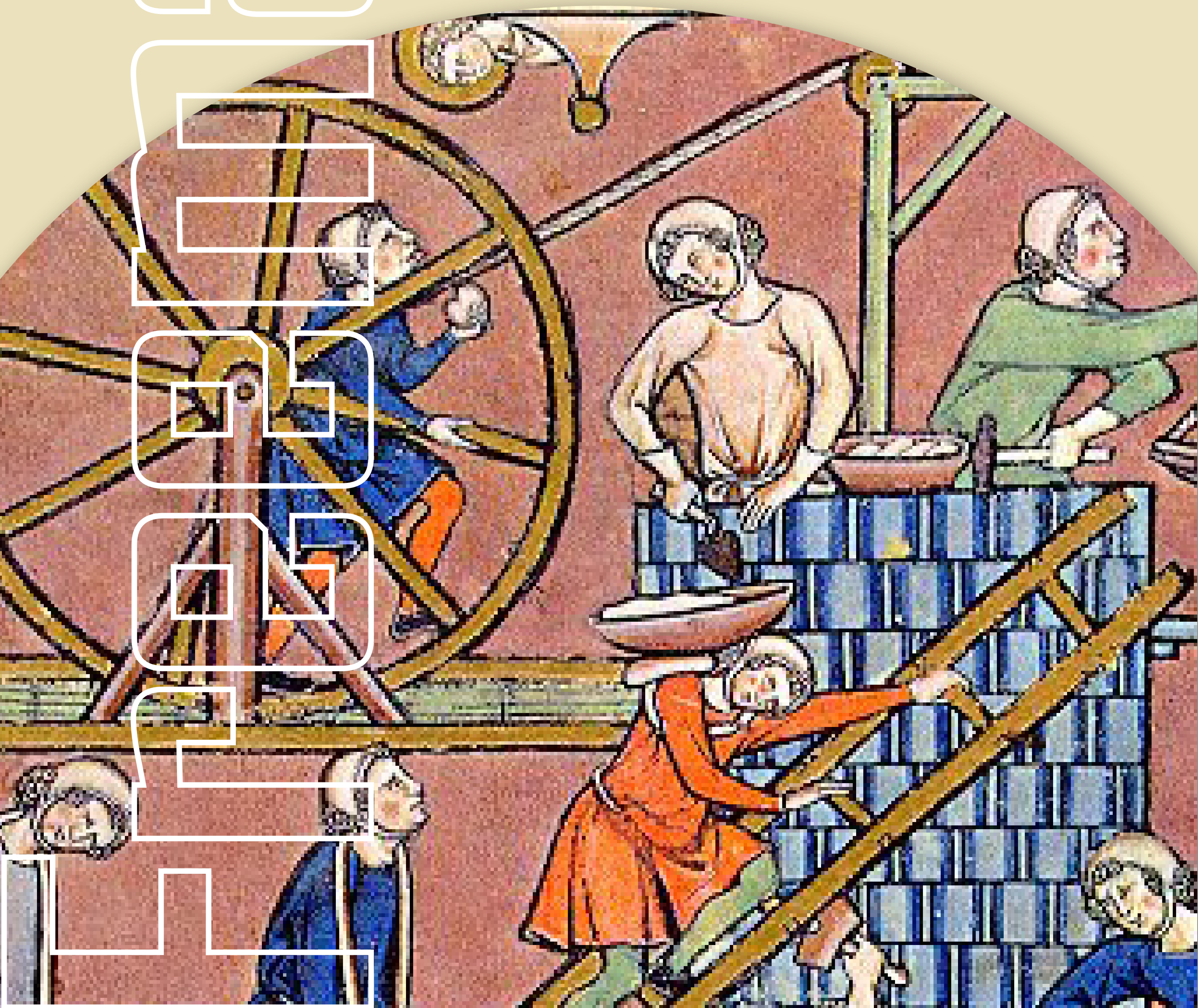
The Freemason Build System

freemason



RADVISION
Delivering the Visual Experience

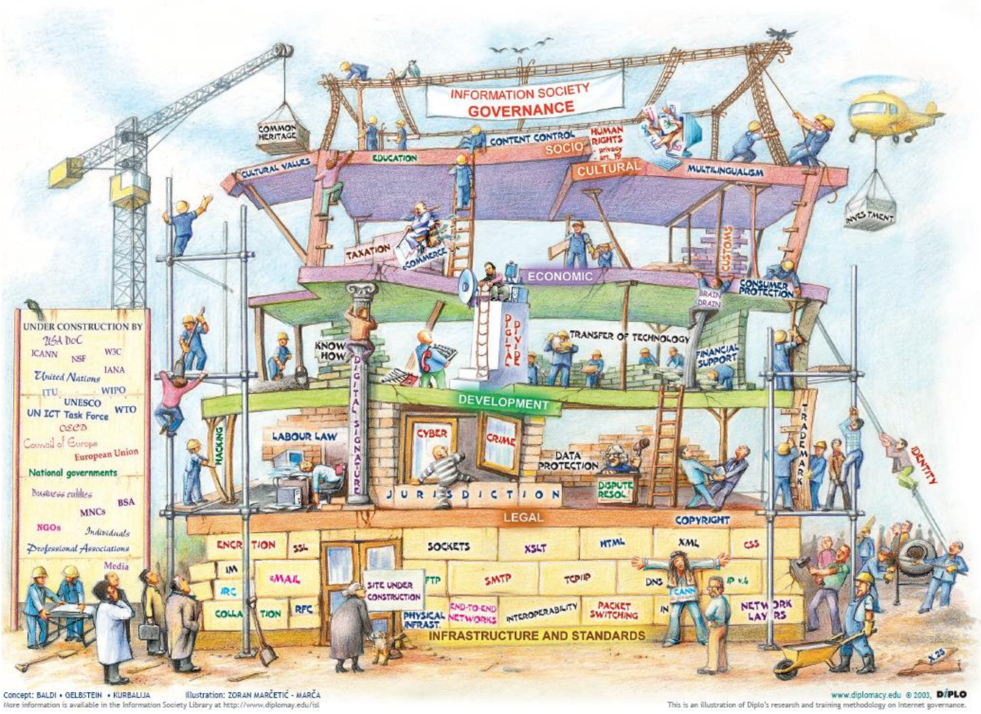
The *Freemason* Build System

Version 4.0

User Guide

version 1.0

July 2, 2007



Then they said, "Come, let us build ourselves a city, with a tower that reaches to the heavens, so that we may make a name for ourselves and not be scattered over the face of the whole earth."

– Genesis 11:4

Contents

Part I

Getting Started with *Freemason*

Chapter 1

An Introduction

Freemason is a build system. Its purpose is to build multi-platform software products from source code, in a coherent and a concise way, by a single, simple command, with no user intervention. One typical use of such facility is within a continuous integration system, which enables automatic software build and testing to take place.
While being biased towards C/C++ projects, it can be used to compile any source code that has 1-to-1 source-to-object relations, such as Java or C#. In addition, it can be used to perform operation that span source files and modules.
In this manual, we first go over the basic features of *Freemason* by reviewing a sample project. Then we'll examine *Freemason*'s architecture and study its features more thoroughly.

Chapter 2

Getting Started

Let's take a look at a simple project, *Sample1*, containing a program and a library (Figures 2.1 and 2.2).

Figure 2.1: Sample1 project

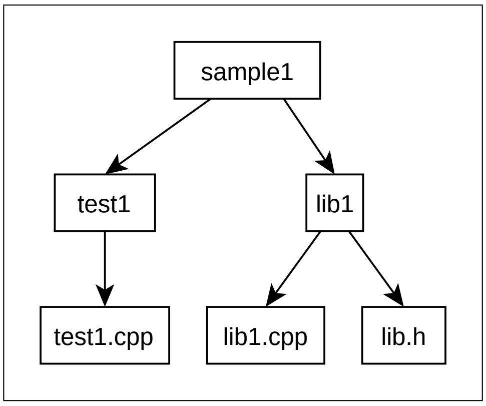


Figure 2.2: Sample1 project sources

`test1/test1.cpp`

```
#include "lib1/lib1.h"

#include <stdio.h>

int main()
{
    printf("%s\n", foo());
    return 0;
}
```

`lib1/lib1.cpp`

```
const char *foo()
{
    return "Hello, World!";
}
```

`lib1/lib1.h`

```
const char *foo();
```

2.1 Makefiles

The first thing we will like to do is to build the code in ***test1*** and ***lib1***. We would like to build the source files in ***lib1*** into a library, and the source files in ***test1*** into an executable file. *Freemason* uses the term ***module*** to refer to a set of source files (like ***test1*** and ***lib1***) that eventually can be compiled into binary form (that is, a library of some sort or an executable, which we call ***products***).

For *Freemason* to build a module it needs some instructions. Those instructions are written in a language that has no name, but it is known to its acquaintances as the language of a bloke called *GNU make*. This file is called, not surprisingly, *makefile*. The makefiles are placed along with the source files (Figure 2.4).

Let's examine the makefile of ***test1*** (Figure 2.3). We first define a variable called `SRC_ROOT` (line 1) that serves as an anchor to the root of our source code. Since ***test1*** resides under ***sample1***, we go two levels up the directory structure to meet the root.

The next line introduces *Freemason*. We will assume here that the *Freemason* definitions files (also called *Framework*) are in `z:/freemason`. However, in the most common (and most recommended) configuration, the *Freemason Framework* is a of the source view. More on that later. Thus, *Freemason* is introduced by including the file `main` from the *Freemason Framework*. Once it is done, the variable `MK` can be used to reference *Freemason* resources.

After we shook hands and got off the coats, it's time to introduce ourselves to *Freemason*. By defining variable `MODULE_NAME` (line 4), an arbitrary text string we set the name of the module. The name of the module should be unique in the sense that no two modules that are used in the same context should have the same name. *Freemason* also uses the module name to name the module's product file (the file that is created by the build process).

Now, before we can haggle over the price, we should establish what we are[1]. The variable `PRODUCT` (line 5) does just that: it tells *Freemason* what will become out of the module once it is compiled. Among the possible values are `prog` (an .exe file or the like), `lib` (.lib or .a file), and `so` (.dll or .so file).

In lines 10 and 27 we introduce the rest of the *Framework* code: the definitions and the rules. Both are concepts rooted in *GNU make* terminology. Since the implementation of the *Framework* is out of our scope, we will not discuss them here. What we need to know for now is that definition should appear before the rules, and that some variables (the ones we just discussed) should appear before *Freemason* definitions are included.

The definitions in lines 14-23 provide information related to preprocessor definitions (`CC_PP_DEFS.common`), include files directories (`CC_INCLUDE_DIRS.common`), and C/C++ source files to be compiled (`CC_SRC_FILES.common`). There are some things to notice here.

1. The `CC` prefix. This implies that we supply information for a C/C++ compiler abstraction, rather than to a specific compiler. Therefore, we see no place where proprietary compiler option can be specified. This might look strange in first sight, but bear in mind that we're dealing with a cross-platform, cross-compiler, cross-my-heart environment. We'll discuss ways of customizing compiler operation in chapters to come.

2. The `.common` suffix. That's to say "for all platforms".

3. The header files of the module. One should not specify header files in `CC_SRC_FILES`. *Freemason* automatically keeps track of header files that are used by source files.

4. The include file directory specified for ***test1***. The directory `"`..” needed in order to allow `"`test1.cpp” to use the header file `lib1/lib1.h`. Using module's name as part of the header file name is considered a good practice.

The makefile of ***lib1*** is very similar. One noticeable difference is the `PRODUCT`, defined to be `lib`.

[1]According to George Bernard Shaw.

Figure 2.3: sample1 makefiles

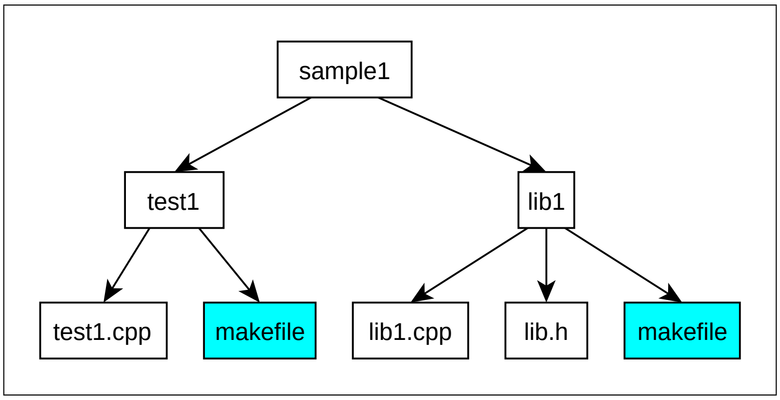
test1/makefile

```
SRC_ROOT=../..
include z:/freemason/4/main

MODULE_NAME=test1

PRODUCT=prog

#-----------------------------

include $(MK)/defs

#-----------------------------

define CC_PP_DEFS.common
endef

define CC_INCLUDE_DIRS.common
    ..
endef

define CC_SRC_FILES.common
    test1.cpp
endef

#-----------------------------

include $(MK)/rules
```

lib1/makefile

```
SRC_ROOT=../..
include z:/freemason/4/main

MODULE_NAME=lib1

PRODUCT=lib

#----------------------------

include $(MK)/defs

#----------------------------

define CC_PP_DEFS.common
endef

define CC_INCLUDE_DIRS.common
endef

define CC_SRC_FILES.common
    lib1.cpp
endef

#----------------------------

include $(MK)/rules
```

Figure 2.4: Sample1 project with makefiles



2.2 Compilation

After we're done with writing makefiles, we can turn to compilation. *Freemason* services are provided through a single command (also called "driver"): **mk**.

There are two things *Freemason* has to know before it can build a C/C++ project:

1. What is the platform for which we want to build (`PLATFORM` variable),
2. Whether we're interested in a debug build (`DEBUG=1`) or an optimized one (`OPT=1`).

For the time being, we'll perform a debug build for a platform called `win32`, that targets x86 machines running Windows, and uses a C++ compiler from Microsoft.

And so, from the `sample1/lib1` directory, we issue the command `mk PLATFORM=win32 DEBUG=1`. *Freemason* responds with:

Figure 2.5: sample1/lib1 win32 compilation

```
Build of lib1 ...

Compiling lib1.cpp ...
Creating library bin/win32-debug/lib1.lib ...
Done.
```

Looks like everything is in order here.

Now, from the `sample1/test1` directory, we issue the command `mk PLATFORM=win32 DEBUG=1`.

Figure 2.6: sample1/test1 win32 compilation (with errors)

```
Build of test1 ...

Compiling test1.cpp ...
Creating program bin/win32-debug/test1.exe ...
make: *** [bin/win32-debug/test1.exe] Error 96
```

Errors. Damn software. Cannot even compile a 5-line project. And it doesn't print a proper error message! Hang the DJ! Hey, wait a minute. There's a `make.log` file (Figure 2.7). Maybe we can find something here.

Figure 2.7: sample1/test1 win32 compilation errors (make.log)

```
--------------------------------------------------------------------------------
Fri Jun 13 06:66:00 JDT 2007
Build of test1 ...
--------------------------------------------------------------------------------
Compiling test1.cpp ...
test1.cpp
test1.obj : error LNK2001: unresolved external symbol
                           "char * __cdecl foo(void)" (?foo@@YAPADXZ)
bin/win32-debug/test1.exe : fatal error LNK1120: 1 unresolved externals
```

As it appears, we've got ourselves an undefined symbol. Obviously, it is the function `foo()`, defined in ***lib1***. Since we didn't tell ***test1*** anything about the latter, it doesn't add the library to its link command, leaving the symbol `foo` undefined.

We have to find a way of telling ***test1*** about ***lib1***. This is done in two stages. First, we establish a definitions file (`defs.mk`, Figure 2.8) that describes the interface of ***lib1*** towards other modules.

Line 1 defines the module's name (corresponds to the one in `makefile`). Line 2 defines the module location, relative to the root of the project, as defined by the `SRC_ROOT` in the makefiles. *Freemason* defines a variable, `VROOT` (*view root*), to point to that location. It implies that all projects in a given context should have the same root. Line 6 defines the module product type, which corresponds to the `PRODUCT` in `makefile`. Last but not least, Lines 4 and 8 introduce *Freemason* framework resources related to module definitions. One may ask, where the variables `MK` and `VROOT` get their values from. The answer comes from the fact that `defs.mk` aren't invoked directly as main makefiles, but rather included by the *Freemason* framework once those variables have been defined.

Figure 2.8: sample1/lib1/defs.mk

```
1 MODULE=lib1
2 MODULE_DIR=$(VROOT)/sample1/lib1
3
4 include $(MK)/module/start
5
6 MODULE_PRODUCT=lib
7
8 include $(MK)/module/end
```

Then, we specify the modules on which ***test1*** depends, with `depends.mk` file (Figure 2.9). Note that in order for a module to be a dependency of another, it has to own a `defs.mk` file. The dependencies file defines the variable `MODULE_DEPENDS.common`, which is a sequence of expressions of the form `(module-name,number-path)`, similar to those that appear in the corresponding `defs.mk` files.

Figure 2.9: sample1/test1/depends.mk

```
define MODULE_DEPENDS.common
    (lib1,$(VROOT)/sample1/lib1)
endef
```

After establishing the `defs.mk` and `depends.mk` files, we'll try to recompile ***test1***.

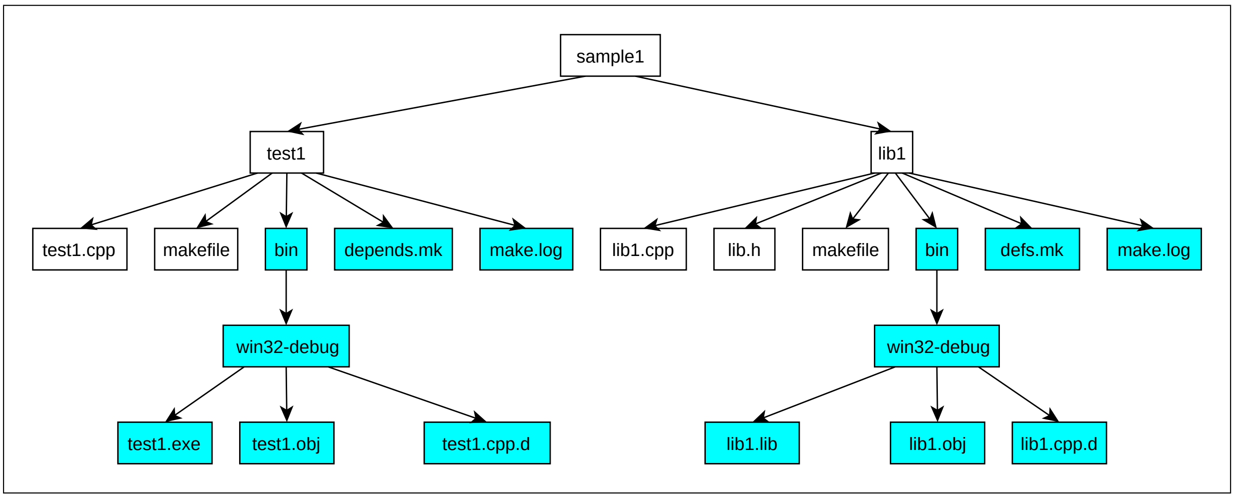
Figure 2.10: sample1/test1 win32 compilation

```
Build of test1 ...

Creating program bin/win32-debug/test1.exe ...
Done.
```

This time, with more luck.

Let's take a look at the ***sample1*** project directories after the build (Figure 2.11).

Figure 2.11: Sample1 project after win32 compilation



A few notes:

1. Modules: A term *module* is central to *Freemason*. It is the basic build unit. We've already seen examples for modules (***test1*** and ***lib1***). For those that seek a formal definition, a module is any location that owns either `makefile` or `defs.mk` files, or both.

2. Dependant modules: modules that are used by other modules called *dependent modules*. The module dependency relation induces a /conceptModule dependency graph. In this example, ***lib1*** is a dependent module of ***test1***.

3. Deep build: If we examine the build process we just performed, we notice that we executed two build commands, one for each module. However, since the modules are dependant, it's logical to expect to build both modules with one command. This can be accomplished by adding `DEEP=1` to the command with which we built module ***test1***. In *Freemason* terms, it's called *Deep Build*, in which a module and all its dependant modules are built.

4. `bin` directory: *Freemason* stores binary files it creates (object files, libraries, executables, etc.) under `bin` directory, followed by a directory that named after the *build variant*, that is, platform and debug/optimization selection. In this example, we compiled for `win32` platform with debug, so the binary files directory is `bin/win32-debug`.

5. `.d` files: *Freemason*'s automatically detects which header files are included by each C/C++ source file it compiles. It then appends `.d` to the name of the source file and stores the results in the binary directory.

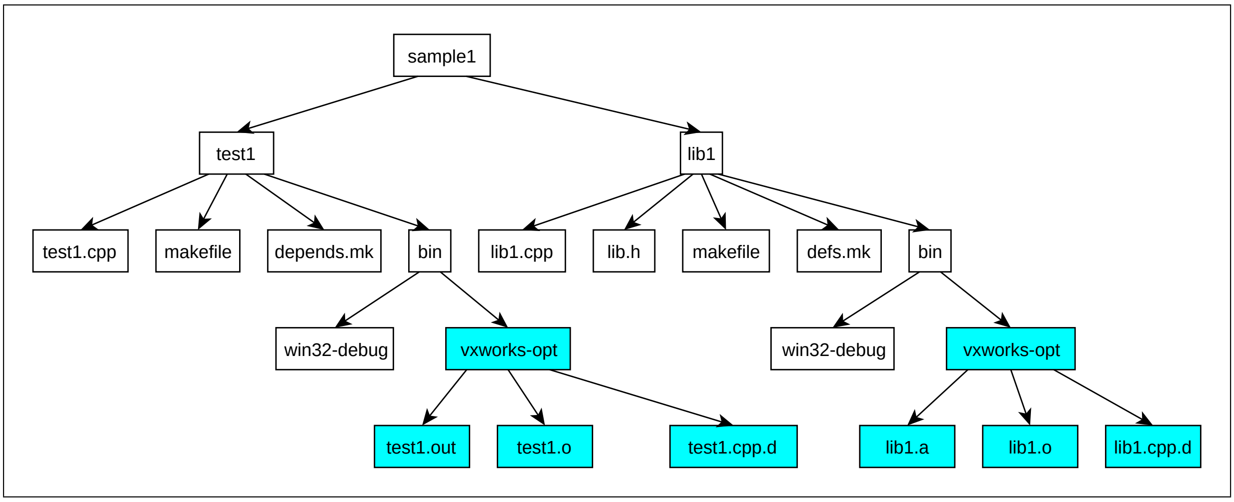
2.3 Yet Another Platform

Once we got hold of building using *Freemason*, we can go on and build for another platform, VxWorks. From `sample1/test1`, we issue the command `mk PLATFORM=vxworks OPT=1 DEEP=1`.
We requested *Freemason* to build an optimized version of ***test1*** and ***lib1*** for VxWorks. Actually, this is a bit misleading, since VxWorks compilations are compiler-dependant and architecture-dependant, therefore there cannot be a generic "vxworks" platform. Thus we assume that our *Freemason* framework contains a proper definition of such architecture, and that is uses a Diab compiler.

Figure 2.12: sample1 compilation for VxWorks

```
Build of lib1 ...

Compiling lib1.cpp ...
Creating library bin/vxworks-opt/lib1.a ...
Done.

Build of test1 ...

Compiling test1.cpp ...
Creating program bin/vxworks-opt/test1.out ...
Done.
```

Figure 2.13: Sample1 project after VxWorks compilation



Examining ***sample1*** directories, we reveal a similar outcome to the Win32 compilation, with some variations: The file extentions of the products is different. Instead of ".exe", ".lib", and ".obj", we find ".out", ".a", and ".o". This is due to platform conventions. *Freemason* is aware of such conventions and names product files accordingly. You can review the makefiles of our products to discover that we didn't specifically mention file extensions anywhere.

2.4 A New, Exciting Feature

Suppose we would like to add a GUI element to our application, say a message box. We will modify ***test1*** in the following mannger (Figure 2.14).

Figure 2.14: test1 with a pretty face

test1/test1.cpp

```
#include "lib1/lib1.h"

#if defined(GUI) && defined(_WIN32)
#include <windows.h>
#endif

#include <stdio.h>

int main()
{
    const char *text = foo();
    printf("%s\n", text);
#if defined(GUI) && defined(_WIN32)
    MessageBox(0, text, "test1", MB_OK);
#endif
    return 0;
}
```

test1/makefile

```
SRC_ROOT=../..
include z:/freemason/4/main

MODULE_NAME=test1

PRODUCT=prog

#-----------------------------

include $(MK)/defs

#-----------------------------

define CC_PP_DEFS.common
endef

define CC_PP_DEFS.windows
    GUI
endef

define CC_INCLUDE_DIRS.common
    ..
endef

define CC_SRC_FILES.common
    test1.cpp
endef

#-----------------------------

include $(MK)/rules
```

Looking at `test1.cpp`, one can notice we use two macros to limit the GUI feature to the Win32 platform: `_WIN32` and `GUI`. The first macro is a predefined in the Microsoft C/C++ compiler, so we don't need to bother about it. The second one is our own way to mark the GUI-related code, so we can turn it on or off at will. We added a `CC_PP_DEFS.windows` definition to the ***test1*** makefile, containing the `GUI` macro. *Freemason* framework knows to look for platform suffixes of variables like `CC_PP_DEFS`, so all we need to do is ask.

2.5 Lib1 Gets a Promotion

One morning, ***lib1*** was notified that she gets promoted. From now on, she is no longer a plain old-fashioned static library, but a shiny, modern DLL. Well, at least for some. It appears that the folks from the VxWorks platform aren't very happy with the last move, and they keep treating ***lib1*** as the good old archive they're used to. And you simply can't do nothing about it. Except, of course, to let *Freemason* deal with it.

But before ***lib1*** takes on its new role, it first needs to clean the desk: its binary directory contains a static library that we don't need anymore. What we need to do, in *Freemason* terms, is a '*clean*' operation: one that removes binary files created by some build operation. As a matter of fact, ***test1***, the sole user of ***lib1*** has to be *clean*-ed too, since it is linked with `lib1.lib`. Although it is enough to merely delete `test1.exe`, we're going for the full clean, for simplicity. And so, from the `sample1/test1` directory, we issue the command

```
mk PLATFORM=win32 DEBUG=1 clean DEEP=1
```

Note that the parameters of `mk` can come in any order. That's right, *Freemason* is the program where everything's made up and the order of the parameters doesn't matter.

Once we're done, we can turn into changing ***lib1*** into a DLL. Thanks to the Microsoft C/C++ cumbersome DLL support, it is much more a C++ programming task than a *Freemason* one. Inspect Figure 2.15 for the source code changes. Notice than it is for the builder of ***lib1*** to define the macro `LIB1_EXPORTS` in order for the potion to work.

Figure 2.15: lib1 dressed like a DLL

lib1/lib1.cpp

```
#include "lib1.h"

LIB1_API const char *foo()
{
    return "Hello, World!";
}
```

lib1/lib1.h

```
#ifdef _WIN32
#    ifdef LIB1_EXPORTS
#        define LIB1_API __declspec(dllexport)
#    else
#        define LIB1_API __declspec(dllimport)
#    endif
#else
#    define LIB1_API
#endif

LIB1_API const char *foo();
```

In the makefiles portion (Figure 2.16), things look peaceful (I thought you should know). Lines 6-7 in `makefile` and 6-7 in `defs.mk` reflect the duality of ***lib1***'s product type in Win32 and VxWorks platforms. Note that `so` stands for *shared object*, a synonym of DLL in the UNIX world. We also added (`makefile`, lines 18-20) a definition for the macro `LIB1_EXPORTS`, in `CC_PP_DEFS.windows`, similarly to what we did for ***lib1*** in the last section. Note that ***test1*** stays intact - we don't need to change it at all. It is still linked with `lib1.lib`, but this time it is ***lib1***'s link library.

Figure 2.16: lib1 Feemason files

lib1/makefile

```
1  SRC_ROOT=..
2  include z:/NBU_BUILD/freemason/4/main
3
4  MODULE_NAME=lib1
5
6  PRODUCT.windows=so
7  PRODUCT.vxworks=lib
8
9  #-----------------------------------
10
11 include $(MK)/defs
12
13 #-----------------------------------
14
15 define CC_PP_DEFS.common
16 endef
17
18 define CC_PP_DEFS.windows
19     LIB1_EXPORTS
20 endef
21
22 define CC_INCLUDE_DIRS.common
23 endef
24
25 define CC_SRC_FILES.common
26     lib1.cpp
27 endef
28
29 #-----------------------------------
30
31 include $(MK)/rules
```

lib1/defs.mk

```
1  MODULE=lib1
2  MODULE_DIR=$(VROOT)/lib1
3
4  include $(MK)/module/start
5
6  MODULE_PRODUCT.windows=so
7  MODULE_PRODUCT.vxworks=lib
8
9  include $(MK)/module/end
```

We next perform the build command `mk PLATFORM=win32 DEBUG=1 DEEP=1` (Figure 2.17) and we're ready to run. If you dare to risk it, you'll find out that running `test1.exe` results in an error, because it cannot find `lib1.dll`, since they reside in different directories.

Figure 2.17: Build of sample1 with lib1 as DLL

```
Build of lib1 ...

Compiling lib1.cpp ...
Creating dynamic library bin/win32-debug/lib1.dll ...
Done.

Build of test1 ...

Compiling test1.cpp ...
Creating program bin/win32-debug/test1.exe ...
Done.
```

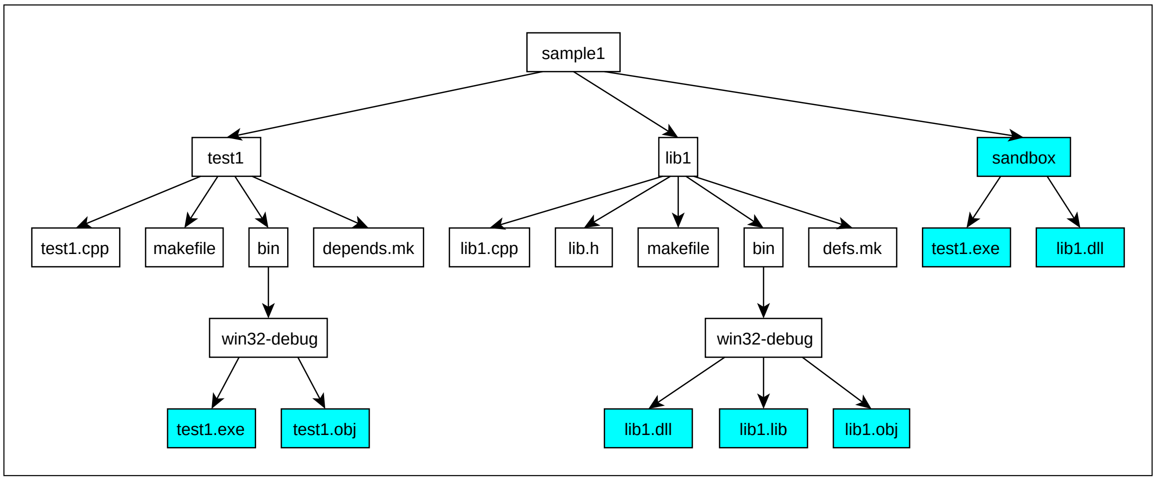
We resolve this problem by asking *Freemason* to *install* the binary files required by `test1.exe` to run. We would like the files to be placed in the directory `sample1/sandbox`, from which we'll run `test1.exe`:

`mk PLATFORM=win32 DEBUG=1 install INSTALL_DIR=../sandbox`

Figure 2.18: Collecting ***sample1*** binaries

```
Installing bin/win32-debug/test1.exe ...
Installing sample1/lib1/bin/win32-debug/lib1.dll ...
Done.
```

Figure 2.19: ***sample1*** project with lib1 as DLL



2.6 New Module in Town

After being promoted, ***lib1*** just had to find itself subordinates. The result of this effort is a new ***strings*** module, shown in Figure 2.20. It is a minimal string class, with makefiles that are more or less identical to the ones of the first ***lib1*** module (recall Figures 2.3 and 2.8). On ***lib1***'s side, we added a `depends.mk` file in order to list ***strings*** as a dependant module, and modified `lib1.cpp` to use the `String` class.

Figure 2.20: ***strings*** module

strings/strings.cpp

```
#include "strings.h"

#include <stdlib.h>

String::String(const char *str)
{
    s = str ? strdup(str) : 0;
}

String::~String()
{
    if (s)
        free(s);
}

const char *String::c_str()
{
    return s ? s : "";
}
```

strings/defs.mk

```
#include <string.h>

class String
{
    char *s;
public:
    String(const char *str = 0);
    ~String();

    const char *c_str();
};
```

strings/makefile

```
SRC_ROOT=..
include z:/NBU_BUILD/freemason/4/main

MODULE_NAME=strings

PRODUCT=lib

#-----------------------------------

include $(MK)/defs

#-----------------------------------

define CC_PP_DEFS.common
endef

define CC_INCLUDE_DIRS.common
endef

define CC_SRC_FILES.common
    strings.cpp
endef

#-----------------------------------

include $(MK)/rules
```

strings/defs.mk

```
MODULE=strings
MODULE_DIR=$(VROOT)/strings

include $(MK)/module/start

MODULE_PRODUCT=lib

include $(MK)/module/end
```

Figure 2.21: ***lib1*** with ***strings***

lib1/lib1.cpp

```
#include "lib1.h"
#include "strings/strings.h"

LIB1_API String foo()
{
    Strins hello = "Hello, World!";
    return hello.c_str();
}
```

lib1/depends.mk

```
define MODULE_DEPENDS.common
    (strings,$(VROOT)/sample1/strings)
endef
```

What happens at the link level? That depends on the platform.

According to Figure 2.22, On VxWorks, ***lib1*** and ***strings*** are libraries (archives), and both are linked into `test1.out`. On Win32, ***lib1*** is a DLL (actually, an executable). Its link library is linked with `test1.exe`, while `strings.lib` is linked with `lib1.dll`. Note that `test1.exe` never meets `strings.lib`.

Figure 2.22: Link Graphs

VxWorks link Win32 link

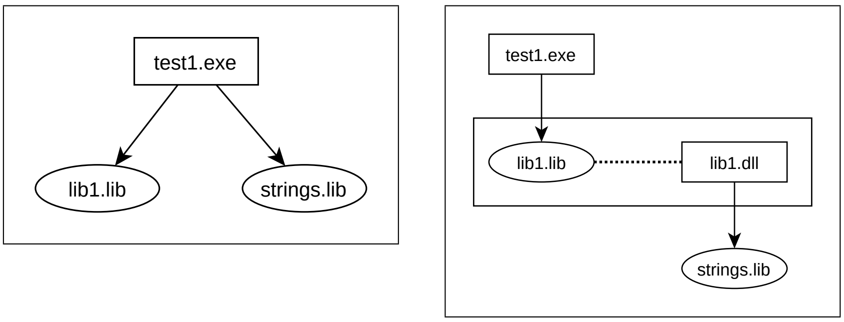
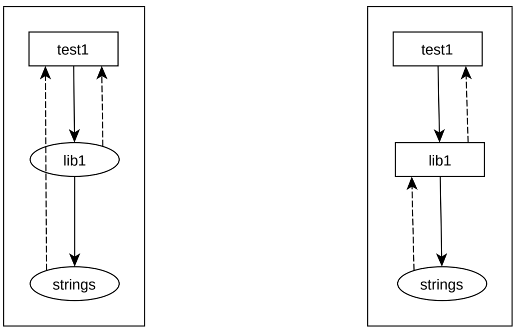


Figure 2.23: ***test1*** Diffusion Graphs

VxWorks diffusion Win32 diffusion



How *Freemason* handles those link scenarios in the most compact way? Through a semantics of *module product diffusion*. Examine the diffusion graphs in Figure 2.23. Notice that at the base of the diffusion graph lies a common module dependency graph, that reflects the relation between symbol definition and usage in terms of modules (that is, if module M1 defined a symbol that is used by module M2, we say that M2 depends on M1).

Now, in the module dependency graph, some modules let products of dependant modules "diffuse" through, while other do not. The quality of diffusion is determined solely by module type. For instance, on VxWorks, a library (***lib1***) will let another library (***strings***) diffuse into a program (***test1***), where it would be linked. On the other hand, on Win32, a DLL (***lib1***) will not allow a library (***strings***) to diffuse through, and link it by itself.

This is the point to mention that we can request *Freemason* to print dependency graphs and diffusion graphs. Due to the potential complexity of such graphs, *Freemason* prints them as trees (by repeating nodes where needed).

Printing a dependency graph is done with the command

```
mk PLATFORM=win32 DEBUG=1 show-deps
```

(`PLATFORM=vxworks` yields the same result)

Figure 2.24: ***test1*** Dependencies and Diffusion Report

Dependencies

```
1 test1 []
2 ... lib1 [test1]
3 ... ... strings [lib1]
```

Win32 Diffusion

```
1 test1 []
2 ... lib1 [test1]
3 ... ... strings [lib1]
4         >> LIB: strings.lib
5     >  LIB: strings.lib
6     >> SO: lib1.lib
7 >> SO: lib1.lib
```

VxWorks Diffusion

```
1 test1 []
2 ... lib1 [test1]
3 ... ... strings [lib1]
4         >> LIB: strings.a
5     >  LIB: strings.a
6     >> LIB: lib1.a strings.a
7 >> LIB: lib1.a strings.a
```

A few words about the reports on Figure 2.24:

1. Module that appears in square brackets is the parent module of the one listed to their left.
2. Lines that begin with `>` list modules that are trying to diffuse through the module on the same level of indentation. For instance, line 5 tells us that the library `strings.lib` is trying to diffuse through module ***lib1***.
3. Lines that begin with `>>` list modules that have successfully diffused through the module on the same level of indentation. For instance, line 6 tells us that is Win32, the link library `lib1.lib` has diffused through module ***lib1***, whereas in VxWorks, both `lib1.a` and `strings.a` have managed to do the same.
4. The last line in the diffusion reports tells us what products will take part of the build process of the root module (in this case, "test1").

Part II

Digging Deeper, Building Higher

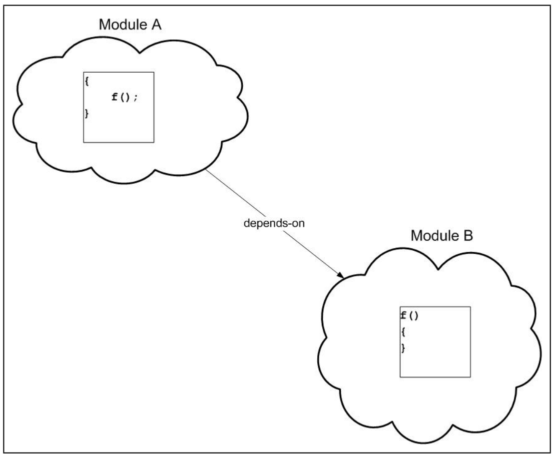
Chapter 3

Freemason Architecture

The *Freemason* world is composed of source code organized into *modules*, each of which containing source files to be built into a *product* (mostly a binary file such as a library or an executable). Products may be built for numerous *targets* (a general name for operating systems and hardware architectures - CPU, board, etc.) using a selection of *tools* (i.e., compilers). In order to reduce complexity of target specification, we use an arbitrary concept of *platform* to denote a particular selection of target OS, target architecture, and build tools. So, in order for *Freemason* to get a clear notion of what is expected from it, one needs only specify the target platform.

Apart from its source files, a module may require the products of other modules in order to build its own. We denote such modules *dependant modules* or just *module dependencies* (not to be confused with dependencies of C/C++ source files on header files, to which we refer as *source file dependencies*). For example, if module A uses a function f() that is defined by module B (Figure ?), we say that module A depends on module B. If products of dependant modules are not available in advance, they should be built before they are required by their dependee. This operation has a recursive manner, since dependant modules may have dependencies of their own. *Freemason* allows modules to specify their dependencies and efficiently takes care of the recursive build operation in a process called *deep build*.

Figure 3.1: Module dependencies



3.1 Framework

Freemason is essentially a combination of makefiles, organized in a hierarchical structure. This collection of makefiles is denoted *Freemason Framework* or simply *Framework*. The actual framework files that are used during a Freemason session depend on the session configuration (that is, values of configuration variables, such as PLATFORM or DEBUG). The framework is of a dynamic nature, and may change from time to time. In the most common scenario, the framework is a treated part of the source code it used to compile, and therefore holds the same version control markers (i.e. labels) as the source code.

3.2 View

Freemason assumes that the file system containing source files in a given build session share a common root directory. The directory structure is called *view*. The common root directory is called *view root*. The only exceptions to this rule are SDK files and the *Freemason* Framework files. It is highly recommended to keep the *Freemason* Framework inside the view, especially when a source control system is involved.

3.3 Repository

In order for *Freemason* to be able to perform actual compilations, it needs to be able to access build tools (i.e. compilers) and SDK resources (header files and libraries). Although it is possible to configure *Freemason* to use such resources from the host that runs the compilation, it is sometimes easier to rely on a common, stable resources. Such resource are stored in a location that's called *Repository*, typically in a shared network location. Since such resources are not always subject to version control, it should contain explicit version specification as part of the directory structure, and avoid modification of the standard, "off-the-self" packages of the development tools. *Freemason* uses the NBU_BUILD_ROOT environment variable to locate its repository.

3.4 GNU make

The engine that drives *Freemason* is GNU make with syntax extension. Thus, the definition files of *Freemason* are actually makefiles. The basic GNU make syntax is valid in *Freemason* files, including variable definitions, conditional control structures and 'include' directives. See also appendix ? for detailed description of syntax additions and changes to the standard GNU make language.

3.5 Software Modules

A *Freemason* module is either a white-box, a black-box, or both. A black-box module is one that ¨knows¨how to build itself from its source files. A black-box module is one that can be referenced and used by other modules (i.e., a module that represents a library that can be linked to an application). Both kind of modules may have dependant modules.
White-box modules own a `makefile` file. Black-box modules own a `defs.mk` file. Module dependencies are described in a `depends.mk` file. Those are the basic *Freemason* module description files. Modules may also have any number of arbitrary makefiles, that may be references (mostly, included) from the basic *Freemason* module description files.
In a typical (and also, recommended) configuration, *Freemason* module description files reside next to the module's source files. However, there's no rule against putting them anywhere in the system, as long as they are kept together. In this case, we should tell *Freemason* how to find the module's source files.

3.6 Environment Variables

Freemason intentionally tries to avoid using of environment variables, to avoid miss-configuration of the build process. An exception to the rule is the variable NBU_BUILD_ROOT, which points to the root of the *Freemason Repository*. All configuration aspects should be handled from within the *Freemason* module configuration files or from the *Framework*.

Chapter 4

Interaction

Freemason services are provided through a single command (also called "driver"): **mk**. All operations concerning a module are performed from the directory containing its definition files. The driver resides at the *Freemason* Repository, and serves as a dispatcher to the GNU make utility in the active view, so the right version is used.

The parameters we supply to the driver divide into three categories:

- Goals: single words, mostly in lowercase. Each goal stands for an action. While it is possible to specify more than one goal in a single command, it is sometimes better to limit ourselves to one goal per command, because different goals may have arguments (variables) with similar names. It is also possible not to specify a goal at all. In this case, the default goal is taken: *Freemason* builds the project.
- Variables: terms of the form VARIABLE=value. Some variables have a general meaning, while others are relate to a specific goal.
- `GNU make` command-line options. See appendix ?.

The parameters may appear in any order.

4.1 Some basic examples

There is one thing *Freemason* has to know before it can build a project: what is the platform for which we want to build. Therefore, we should set the variable `PLATFORM` to one of the following:

- `rv-win32`
- `rv-755`
- `rv-tamar`

As said earlier, there is nothing special about those platforms. They are just a combination of configuration settings. Those names aren't hard-coded anywhere. For instance, if we define a new platform with the same attributes as one of the platforms listed above and build it, the results will be exactly the same as if we used the original one.
For C/C++ projects, another setting is required: whether we require debug information (`DEBUG=1`) or we prefer an optimized build (`OPT=1`). Those settings are mutually exclusive.

Chapter 5

Modules

A module owns three definition files, describing its properties:

- `makefile`
- `defs.mk`
- `depends.mk`

Chapter 6

Configuration

6.1 Product

Supported products include:

prog Windows executable or VxWorks .out/.fls file

lib Library (archive)

so Shared Object (DLL) (Windows only)

pl Partially-linked Object (VxWorks only)

winapp Windows Application

mfcapp Windows MFC Application

none No product (used in products that do nothing but probably get paid quite a lot)

6.2 Target OS

Supported target operating systems include (via `$(TARGET_OS)`):

win32 Windows

vxworks-5.5 VxWorks 5.5

vxworks-6.3 VxWorks 6.3

Target OS types (via `$(TARGET_OS_TYPE)`):

windows Windows

vxworks VxWorks

6.3 Target Platform

Currently defined target platforms:

rv-win32 Windows platform

> **Target OS** Windows
> **Target Architecture** x86
> **CC tool** Microsoft C/C++ Compiler version 12.0

rv-755 RV755 platform

- **Target OS** VxWorks 5.5
- **Target Architecture** RV755 board
- **CC tool** Diab 5.0

rv-tamar TAMAR platform

- **Target OS** VxWorks 6.3
- **Target Architecture** TAMAR board
- **CC tool** Diab 5.4

Chapter 7

Builder Host

Freemason does not require installation of any development tool, IDE or SDK. However, if one wishes that *Freemason* will use such a component, it is easy to establish such configuration without modifying the framework.

Chapter 8

Goals

Chapter 9

The Repository

Chapter 10

Modules Revisited

10.1 Source Files

10.2 Writing Module Definitions Files

10.3 Source file dependencies

10.4 Module dependencies

10.5 Writing Module-Dependency Files

Chapter 11

C/C++ Preprocessing Facilities

Chapter 12

Reference

12.1 Make Targets

cc

cpp

clean

install

show-deps

12.2 Installation

1. Define the `NBU_BUILD_ROOT` environment variable to `r:/build` (we assume `r:` is mapped to `\\storage\\NBU\\Build`
2. Append the directory `%NBU_BUILD_ROOT%\sys\scripts\bin` to the back of your PATH.

Freemason does not require installation of any development tool, IDE or SDK. It is planned to coexist with development tools installed locally on a builder host. It can, however, use the resources on that host if requested. More on that later.

12.3 Variables

DEEP

SHOW_CMD

SHOW_DEPS

SHOW_DIFFUSE

FILE

12.3.1 Useful MAKE options

-n

-jN

-k

-B

-P

-v

12.4 Target Architectures

Currently, the concept of target architecture corresponds to the target system's CPU.

Supported target architectures include:

x86 Standard Intel-based PC

ppc-604 RV755 board

ppc-85xx TAMAR board

12.5 Build Tools

Supported build tools include:

msc-12 Microsoft C/C++ Compiler version 12.0 (i.e. Visual Studio 6)

diab-5.0 Diab 5.0 (i.e. VxWorks 5.5, Tornado 2.2)

diab-5.4 Diab 5.4 (i.e. VxWorks 6.3, Workbench 2.5)

diab-5.5 Diab 5.5 (i.e. VxWorks 6.3, Workbench 2.6)

cc CC (C/C++ compiler abstraction)

asn.1 ASN.1 compiler

Additional tools we plan to support:

- GCC C/C++ Compiler
- Flex and Bison

12.6 Other development tools

Those tools are not categorized as primary build tools, but may take part in the build process or provide useful information during development. They are accessible through specification of a *MAKE target* or a MAKE variable.

Supported tools:

- GNU CPP (C preprocessor)

Additional tools we plan to support:

- PC-Lint
- BoundsChecker, Insure++ (memory debugging tools for Windows)
- BullseyeCoverage (coverage analysis tool)
- ElectricFence, Valgrind (memory debugging tools for Linux)