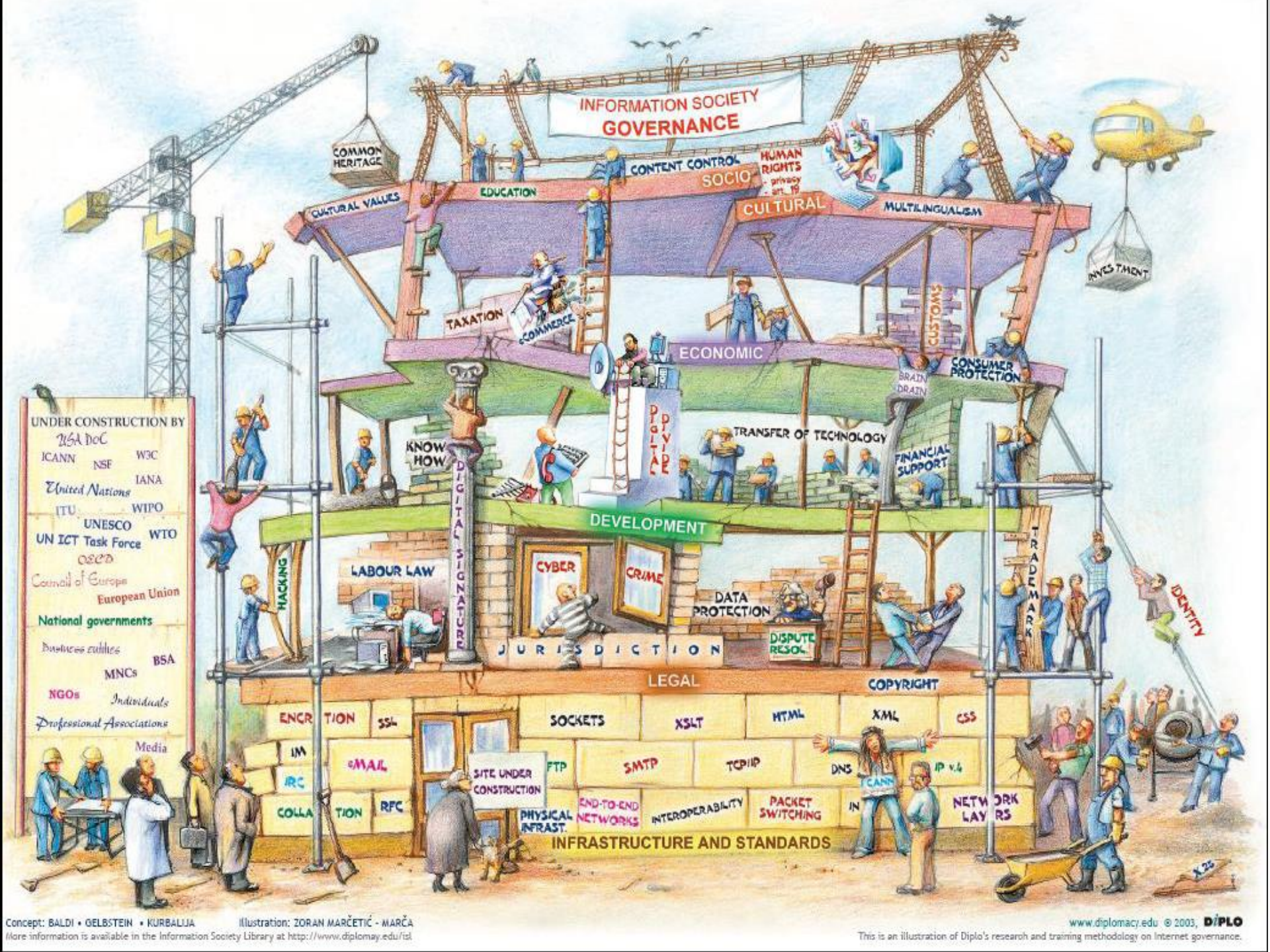
July 2007

The Freemson Build System

Version 4.0

Rafi Einstein

RADVISION®
Delivering the Visual Experience™



Concept: BALDI • GELBSTEIN • KURBALIJA Illustration: ZORAN MARČETIĆ - MARČA

More information is available in the Information Society Library at http://www.diplomacy.edu/isl

This is an illustration of Diplo's research and training methodology on Internet governance.

Freemason

Overview

The Freemason – What is it?

- 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.
- Biased towards C/C++ Projects.
- Separation of concerns: Build vs. Release

Freemason Architecture

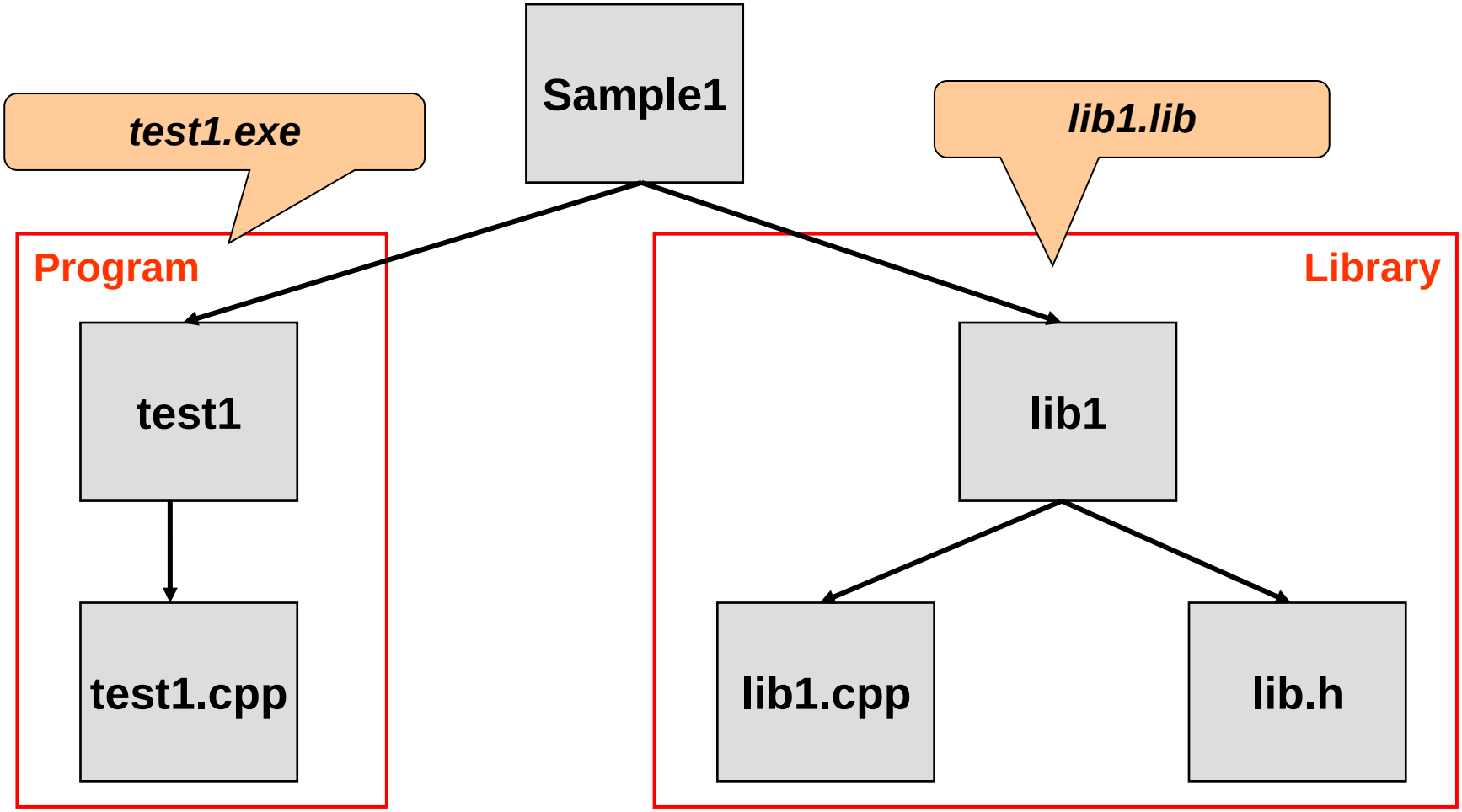
- Framework
  - Hierarchy of makefiles
- Projects
  - In a moment…
- GNU make
  - With many language extensions
- Repository
  - Compilers
  - SDKs
  - Scripts
  - Site Configuration

Freemason

By Example

RADVISION®
Delivering the Visual Experience™

Simple Project: “Sample1”



“Sample1” - Source Code

test1/test1.cpp

```
#include "lib1/lib1.h"
#include <stdio.h>

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

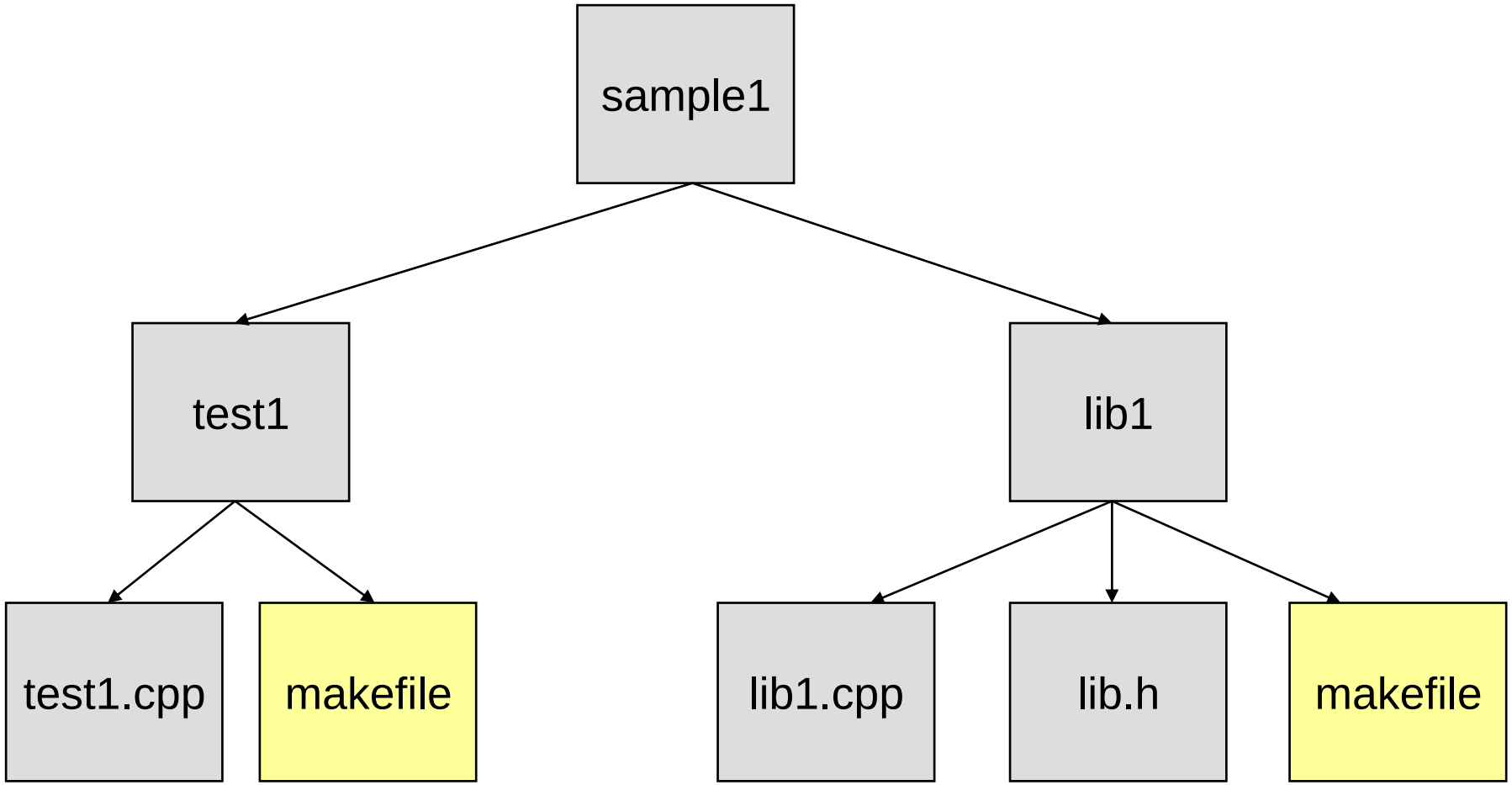
lib1/lib1.cpp

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

lib1/lib1.h

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

Simple Project: “Sample1”



Example2 – Simple Project with Makefile

test1/makefile

```
1   SRC_ROOT=../..
2   include z:/freemason/4/main
3
4   MODULE_NAME=test1
5
6   PRODUCT=prog
7
8   #------------------
9
10  include $(MK)/defs
11
12  #----------------------
13
14  define CC_PP_DEFS.common
15  endef
16
17  define CC_INCLUDE_DIRS.common
18      ..
19  endef
20
21  define CC_SRC_FILES.common
22      test1.cpp
23  endef
24
25  #--------------------------
26
27  include $(MK)/rules
```

lib1/makefile

```
ason/4/main

AME=lib1

lib

----------------------------

include $(MK)/defs

----------------------------

define CC_PP_DEFS.common
endef

define CC_INCLUDE_DIRS.common
endef

define CC_SRC_FILES.common
    lib1.cpp
endef

----------------------------

lude $(MK)/rules
```

An anchor to the root of the source code

Location of the Freemason definition files

An arbitrary text string to set the name of the module

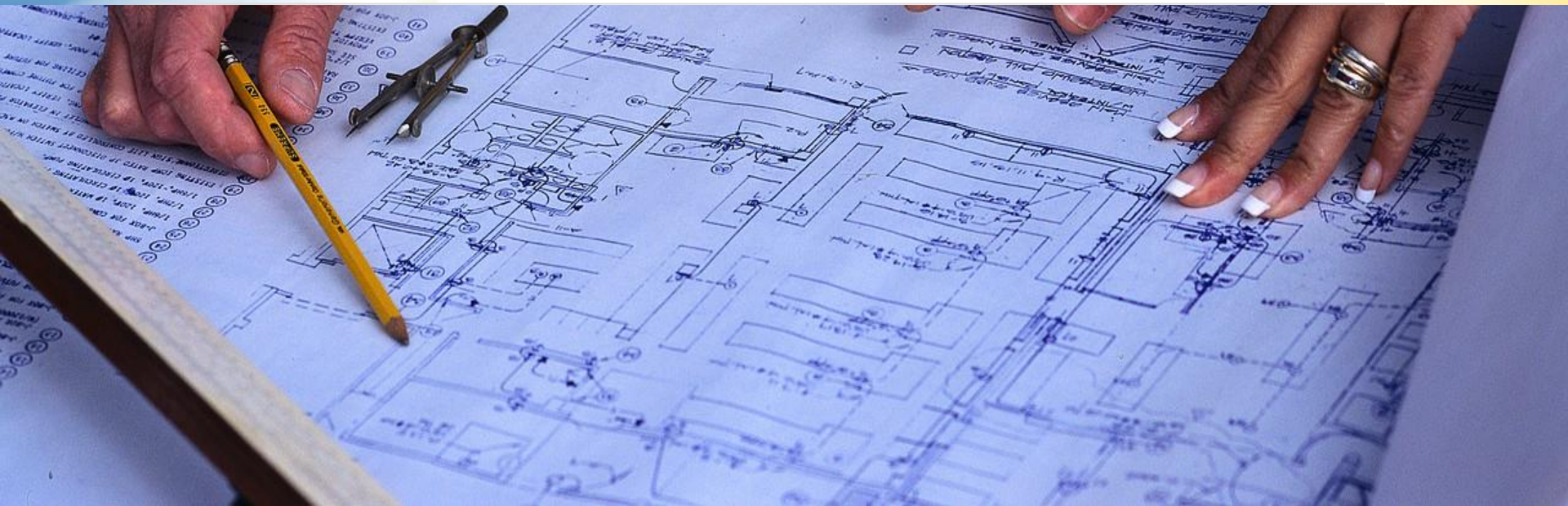
What will become out of the module once it is compiled

Framework code: definitions and rules

Definitions of related processor, include file dirs, and C/C++ source files to be compiled

RADVISION®
Delivering the Visual Experience™

Compiling for Windows



Freemason Compilation Command

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 are interested in a debug build (DEBUG=1) or an optimized one (OPT=1).

- In this part of the lesson, we will perform a debug build for a platform called “win32”, that targets x86 machines running Windows, and uses a C++ compiler from Microsoft.
- Freemason services are provided through a single command (also called ”driver”): **mk**.

Getting Help: `mk help`

```
mk [make-options] [variables...] goal

variables:
  PLATFORM=<platform>     build for selected platform
  OPT=[0|1]               build with optimization
  DEBUG=[0|1]             build without optimization,
                          generate symbolic info
  DEEP=[0|1]              build dependant modules (except
                          prebuilt ones)

  SHOW_CMD=[0|1]          show commands before
                          executing them
  SHOW_DEPS=[0|1]             show module dependency tree
  SHOW_DIFFUSE=[0|1|full]     show diffusion of module
                              products

goals:
  <none>                build the current module
  cc                    compile source file FILE
    FILE=<filename>
  cpp                   preprocess source file FILE,
                      producing .i and .itree files
    FILE=<filename>
  clean                 remove generated files
  install               copy executable files into INSTALL_DIR
    INSTALL_DIR=<dirname>

make options:
  -B                            unconditionally make all
                                targets
  -C DIRECTORY                  change to DIRECTORY before
                                doing anything
  -d                            print lots of debugging
                                information
  -f FILE                       read FILE as a makefile
  -I                            ignore errors from commands
  -j[N]                         allow N jobs at once; infinite
        jobs with no arg
  -k                            keep going when some targets
        can't be made
  -n                            don't actually run any
                                commands; just print them
  -P                            print makefile source, do
        nothing
  -v                            print the version number of
                                make and exit
  -W FILE                       consider FILE to be infinitely
                                new
  --warn-undefined-variables    warn when an undefined
        variable is referenced
```

RADVISION®
Delivering the Visual Experience™

sample1/lib1 Windows Compilation

From the sample1/lib1 directory, we issue the command:

```
mk PLATFORM=win32 DEBUG=1
```

Freemason responds with:

```
1  Build of lib1 ...
2
3  Compiling lib1.cpp ...
4  Creating library bin/win32-debug/lib1.lib ...
5  Done.
```

sample1/test1 Windows Compilation

From the sample1/test1 directory, we issue the command

```
mk PLATFORM=win32 DEBUG=1
```

Freemason responds with:

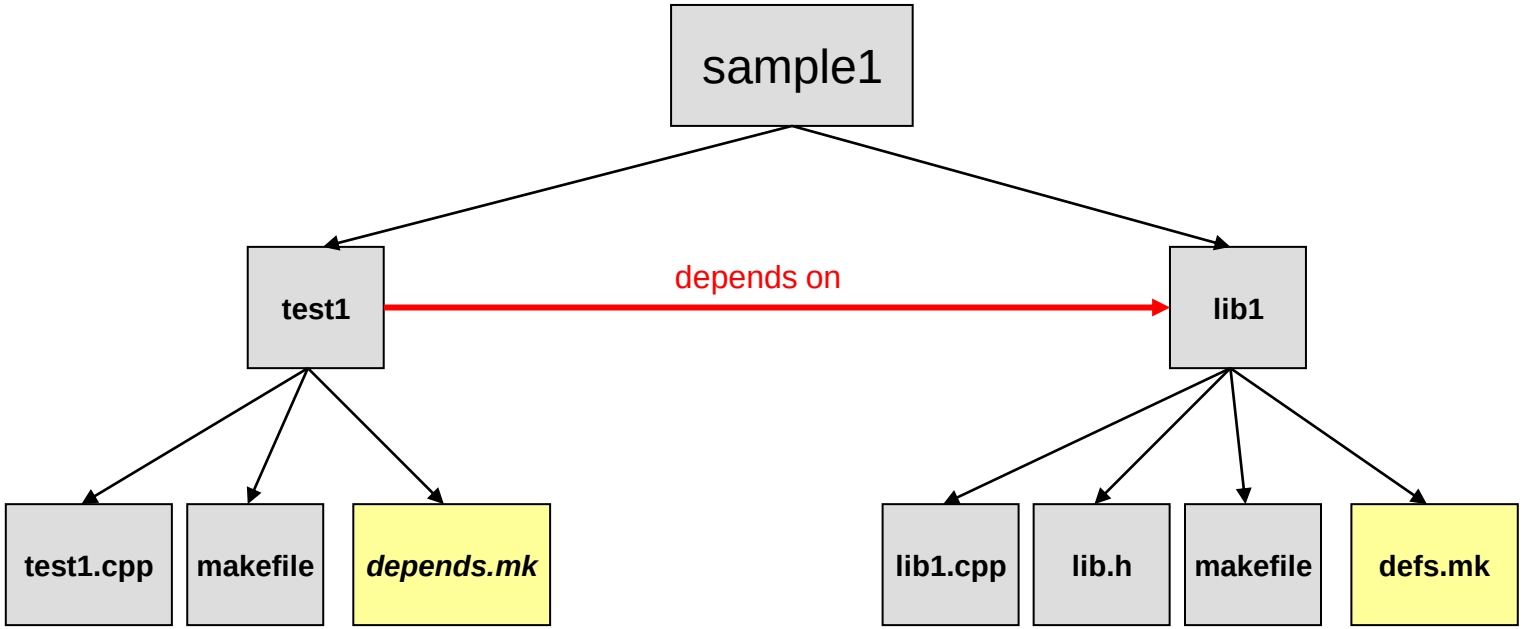
```
1  Build of test1 ...
2
3  Compiling test1.cpp ...
4  Creating program bin/win32-debug/test1.exe ...
5  make: *** [bin/win32-debug/test1.exe] Error 96
```

Compilation Errors

To investigate compilation errors, we inspect the **`make.log`** file, in the sample1/test1 directory:

```
1  ----------------------------------------------------------------
2  Fri Jun 13 06:66:00 JDT 2007
3  Build of test1 ...
4  ----------------------------------------------------------------
5  Compiling test1.cpp ...
6  test1.cpp
7  test1.obj : error LNK2001: unresolved external symbol
8      "char * __cdecl foo(void)" (?foo@@YAPADXZ)
9  bin/win32-debug/test1.exe : fatal error LNK1120: 1 unresolved
10     externals
```

Sample1 Module Dependencies



Module Interface and Dependencies Specification

sample1/test1/depends.mk

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

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
```

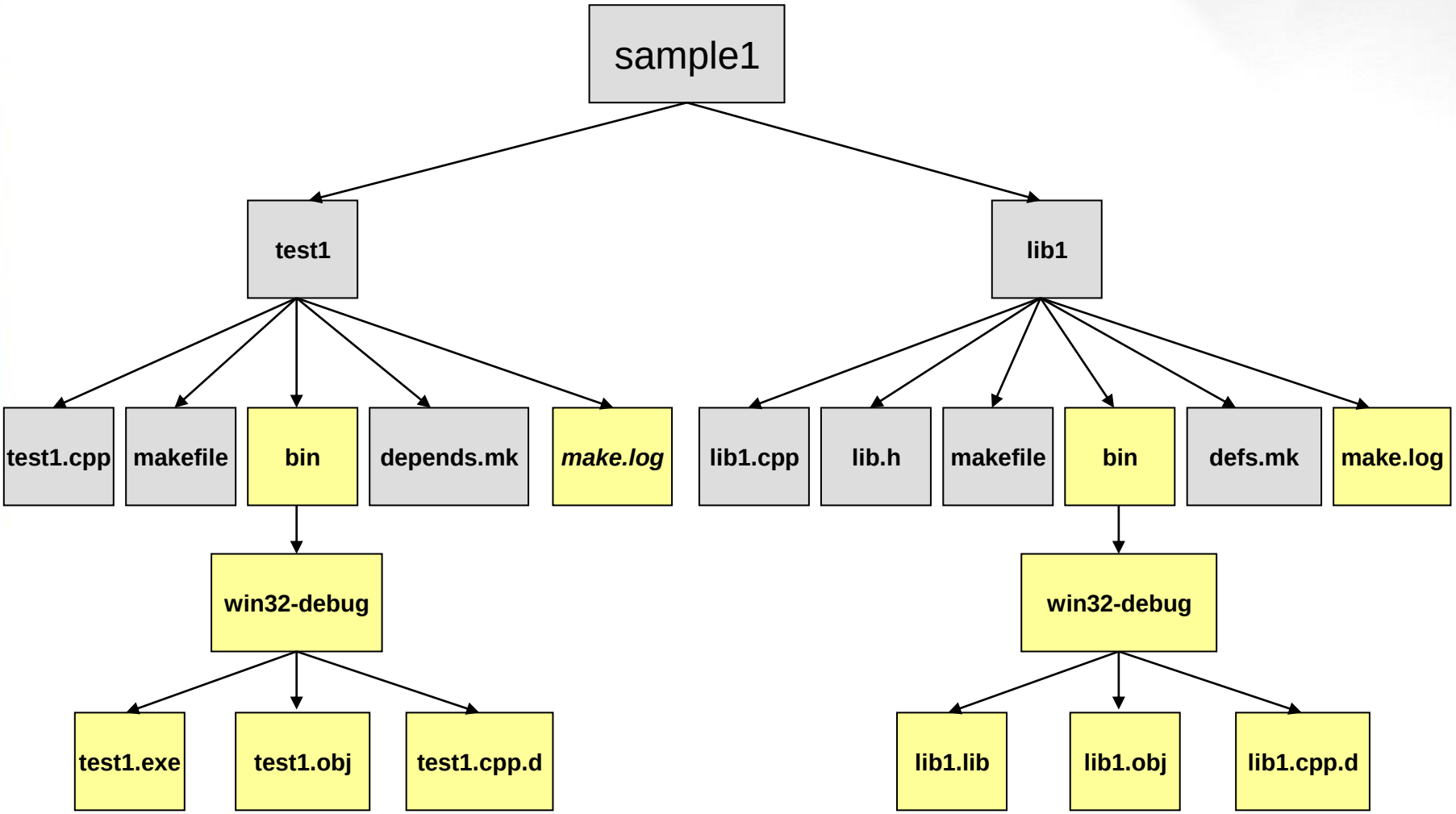
Recompiling sample1/test1

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

sample1/test1 win32 compilation:

```
1 Build of test1 ...
2
3  Creating program bin/win32-debug/test1.exe ...
4  Done.
```

Sample1 Project After Windows Compilation



Summary – Compiling for Windows

- **Modules**
  A term module is central to Freemason. It is the basic build unit.

- **Dependant modules**
  Modules that are used by other modules called dependent modules. The module dependency relation induces a module dependency graph.

- **Deep build**
  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.

Summary – Compiling for Windows (cont.)

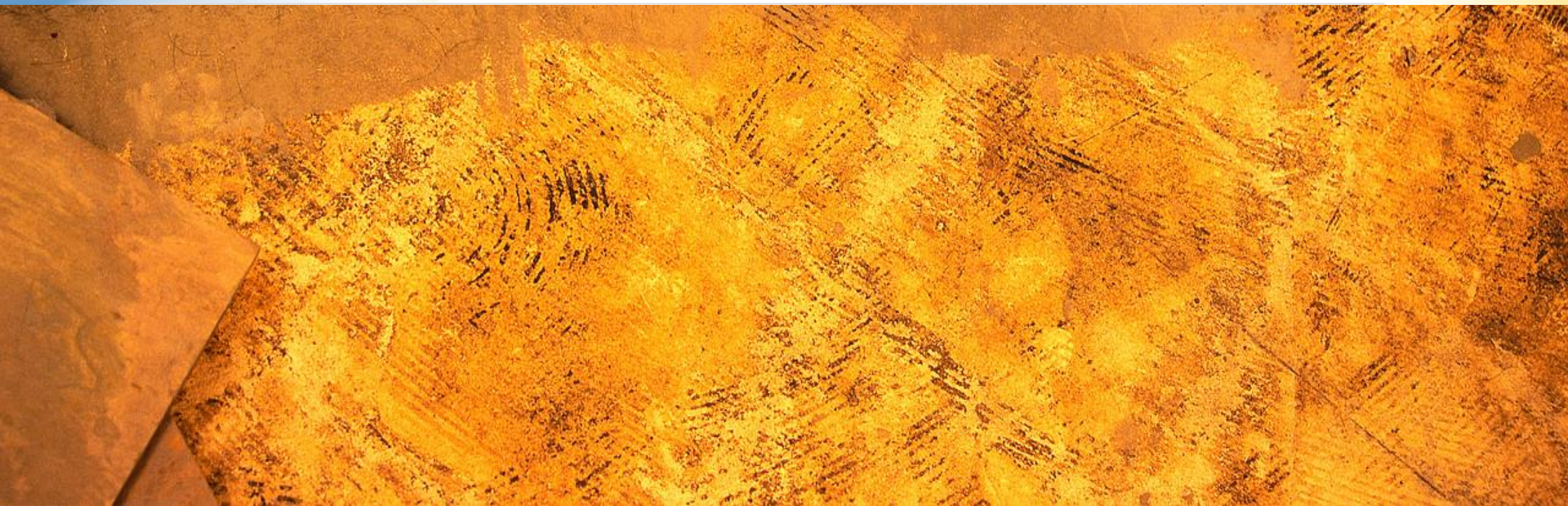
- **bin directory**

  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.

- **.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.

Compiling for VxWorks



Compiling for WxWorks

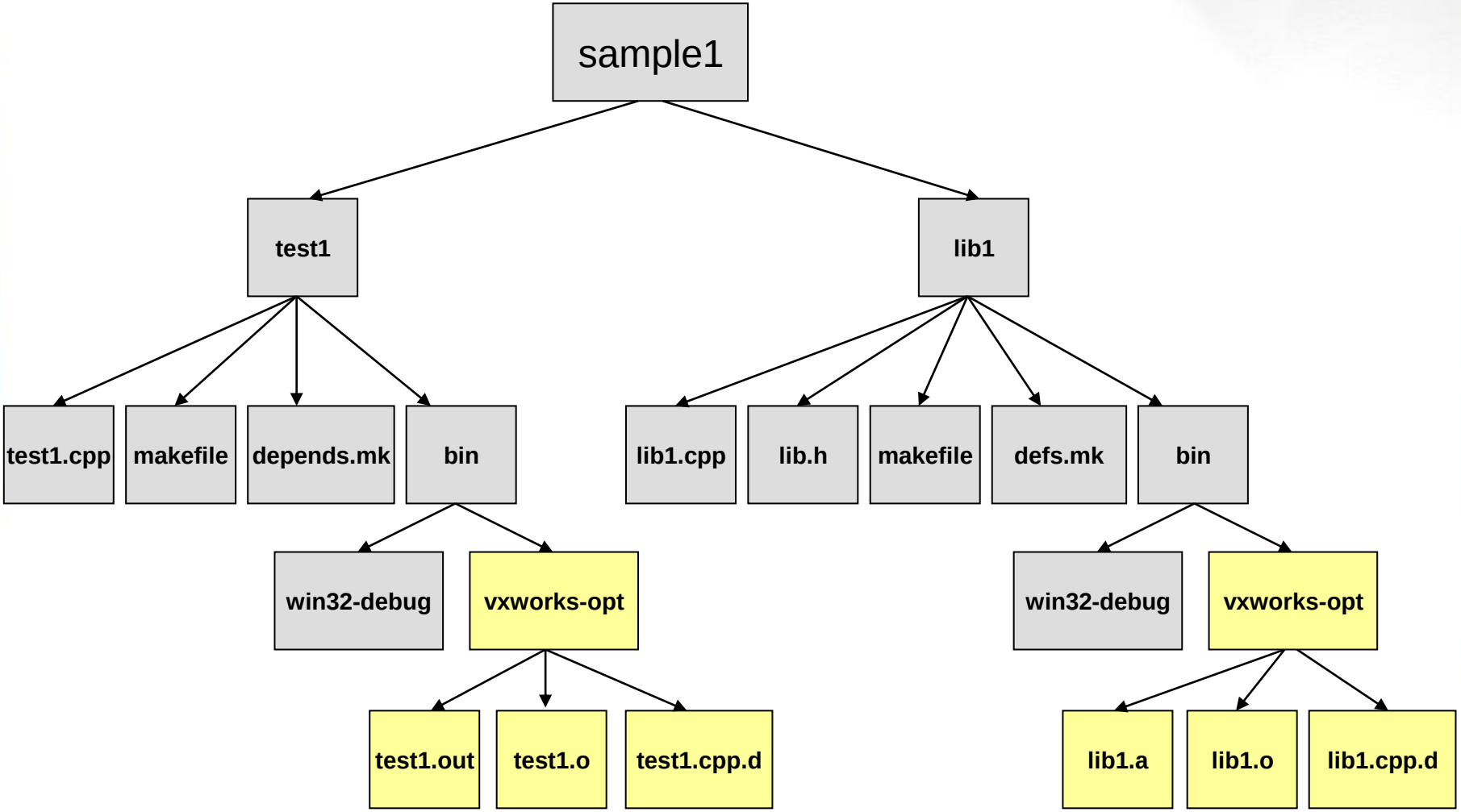
From the sample1/test1 directory, we issue the command

`mk PLATFORM=vxworks OPT=1 DEEP=1`

Sample1 compilation for VxWorks:

```
 1  Build of lib1 ...
 2
 3  Compiling lib1.cpp ...
 4  Creating library bin/vxworks-opt/lib1.a ...
 5  Done.
 6
 7  Build of test1 ...
 8
 9  Compiling test1.cpp ...
10  Creating program bin/vxworks-opt/test1.out ...
11  Done.
```

Sample1 project after VxWorks compilation



Adding GUI Element

Adding a GUI Element to the Application

test1/test1.cpp

```
1  #include "lib1/lib1.h"
2
3  #if defined(GUI) && defined(_WIN32)
4  #include <windows.h>
5  #endif
6
7  #include <stdio.h>
8
9  int main() {
10    const char *text = foo();
11    printf("%s\n", text);
12 #if defined(GUI) && defined(_WIN32)
13    MessageBox(0, text, "test1", MB_OK);
14 #endif
15    return 0;
16 }
17
```

test1/makefile

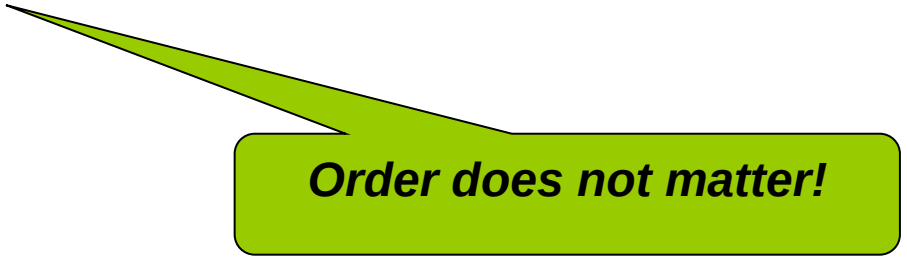
```
1  SRC_ROOT=../..
2  include z:/freemason/4/main
3
4  MODULE_NAME=test1
5
6  PRODUCT=prog
7
8  #----------------------------
9
10 include $(MK)/defs
11
12 #----------------------------
13
14 define CC_PP_DEFS.common
15 endef
16
17 define CC_PP_DEFS.windows
18      GUI
19 endef
20
21 define CC_INCLUDE_DIRS.common
22      ..
23 endef
24
25 define CC_SRC_FILES.common
26      test1.cpp
27 endef
28
29 #----------------------------
30
31 include $(MK)/rules
```

Lib1 Goes Dynamic

Lib1 as a DLL (1)

Preliminary step: cleaning test1

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



Lib1 as a DLL (2)

Turning Lib1 into a DLL:

lib1/lib1.cpp

```
1  #include "lib1.h"
2
3  LIB1_API const char *foo() {
4    return "Hello, World!";
5  }
6
```

lib1/lib1.h

```
1  #ifdef _WIN32
2  #  ifdef LIB1_EXPORTS
3  #     define LIB1_API __declspec(dllexport)
4  #  else
5  #    define LIB1_API __declspec(dllimport)
6  #  endif
7  #else
8  #  define LIB1_API
9  #endif
10
11 LIB1_API const char *foo();
```

Lib1 Freemason 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
```

Build of Sample1 With Lib1 as a DLL

Performing the build command:

mk PLATFORM=win32 DEBUG=1 DEEP=1

```
1  Build of lib1 ...
2
3  Compiling lib1.cpp ...
4  Creating dynamic library bin/win32-debug/lib1.dll ...
5  Done.
6
7  Build of test1 ...
8
9  Compiling test1.cpp ...
10 Creating program bin/win32-debug/test1.exe ...
11 Done.
```

Running Test1 with Lib1 as a DLL

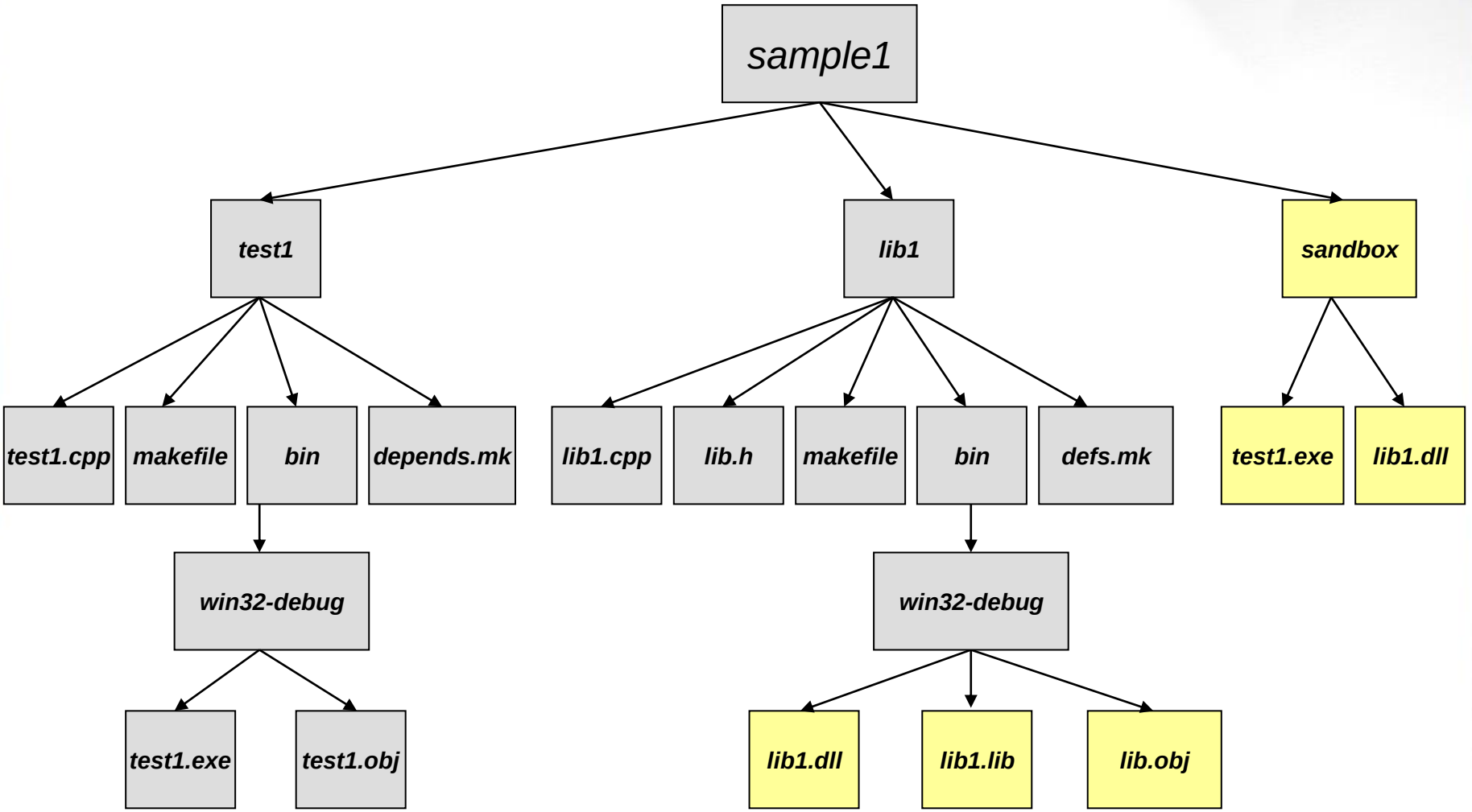
If we try to run test1.exe we get an error, because it cannot find lib1.dll.
We resolve the problem by asking *Freemason* to copy files required by test1.exe to run to a single directory:

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

Collecting Sample1 binaries:

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

Sample1 Project With Lib1 as DLL



Adding A New Module

String Module (1)

strings/strings.cpp

```
1  #include "strings.h"
2
3  #include <stdlib.h>
4
5  String::String(const char *str) {
6    s = str ? strdup(str) : 0;
7  }
8
9  String::~String() {
10   if (s)
11     free(s);
12 }
13
14 const char *String::c_str() {
15   return s ? s : "";
16 }
17
18
19
```

strings/strings.h

```
1  #include <string.h>
2
3  class String {
4    char *s;
5
6  public:
7    String(const char *str = 0);
8    ~String();
9
10   const char *c_str();
11 };
12
```

String Module (2)

strings/makefile

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

strings/defs.mk

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

lib1 With Strings

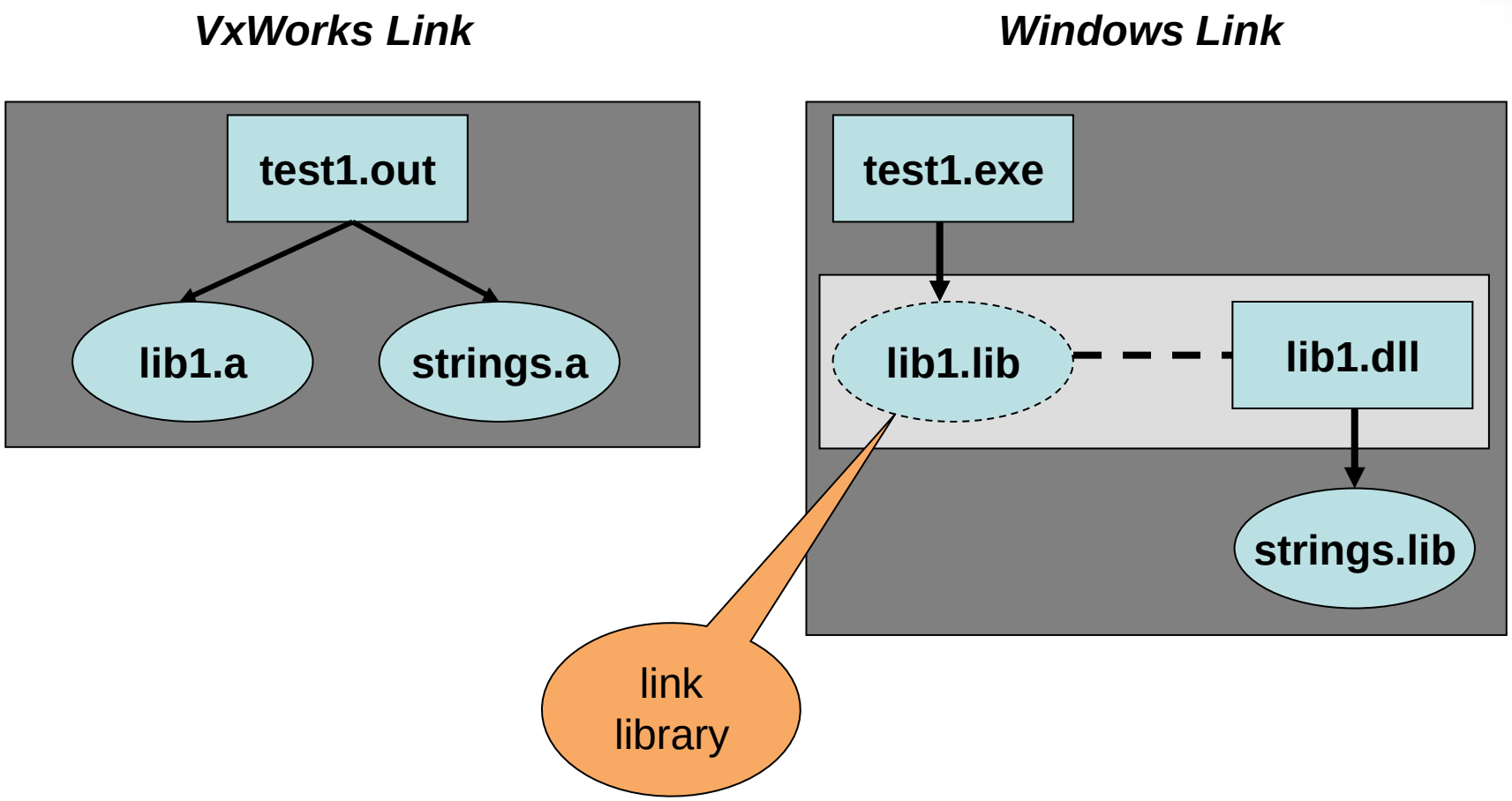
lib1/lib1.cpp

```
1  #include "lib1.h"
2  #include "strings/strings.h"
3
4  LIB1_API String foo() {
5    Strins hello = "Hello, World!";
6    return hello.c_str();
7  }
8
```

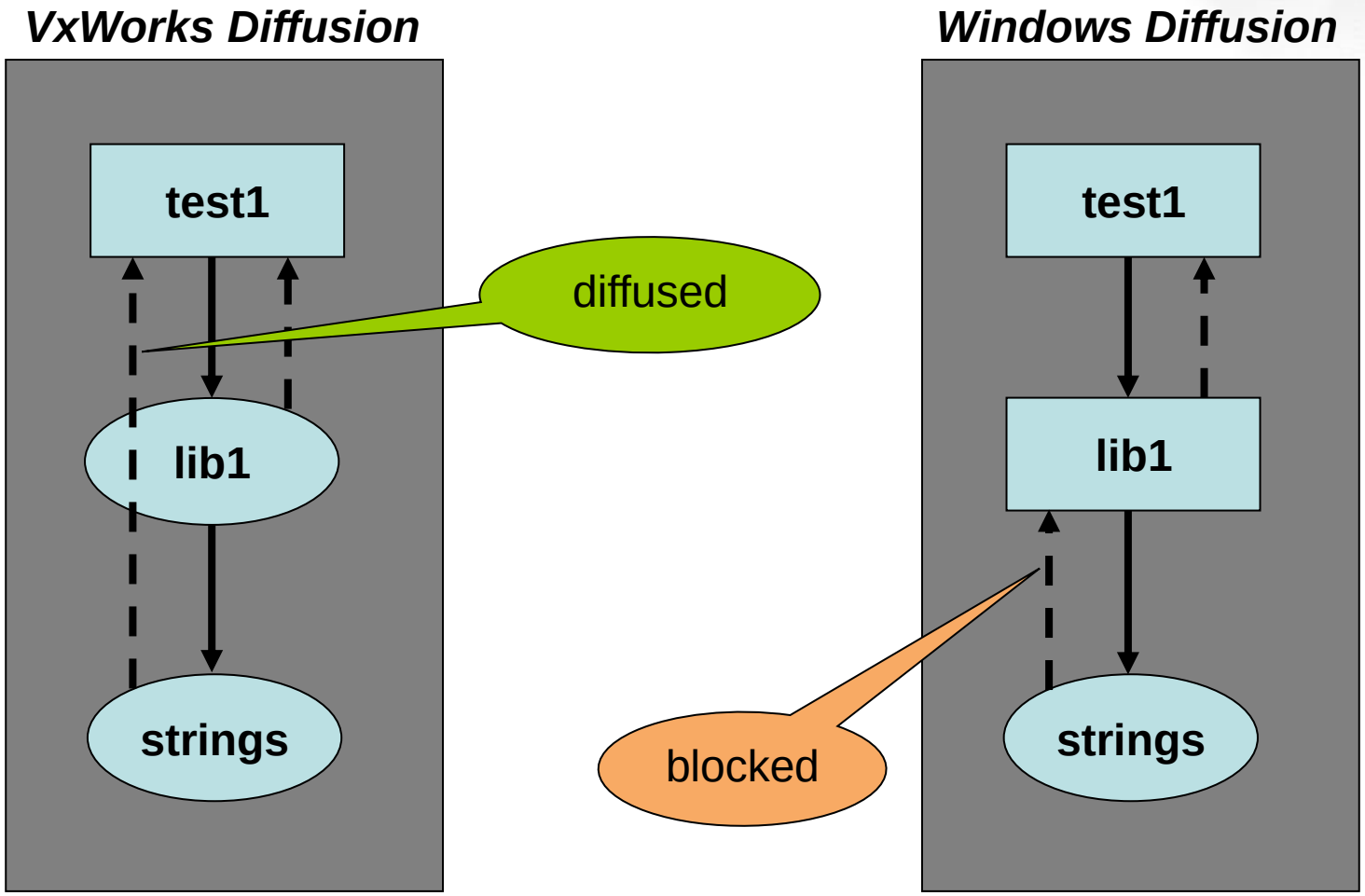
lib1/depends.mk

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

Link Diagrams



test1 Module Diffusion



test1 Dependencies and Diffusion Report

Printing a dependency graph is done with the command:

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

(PLATFORM=vxworks yields the same result)

Dependencies

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

Windows 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

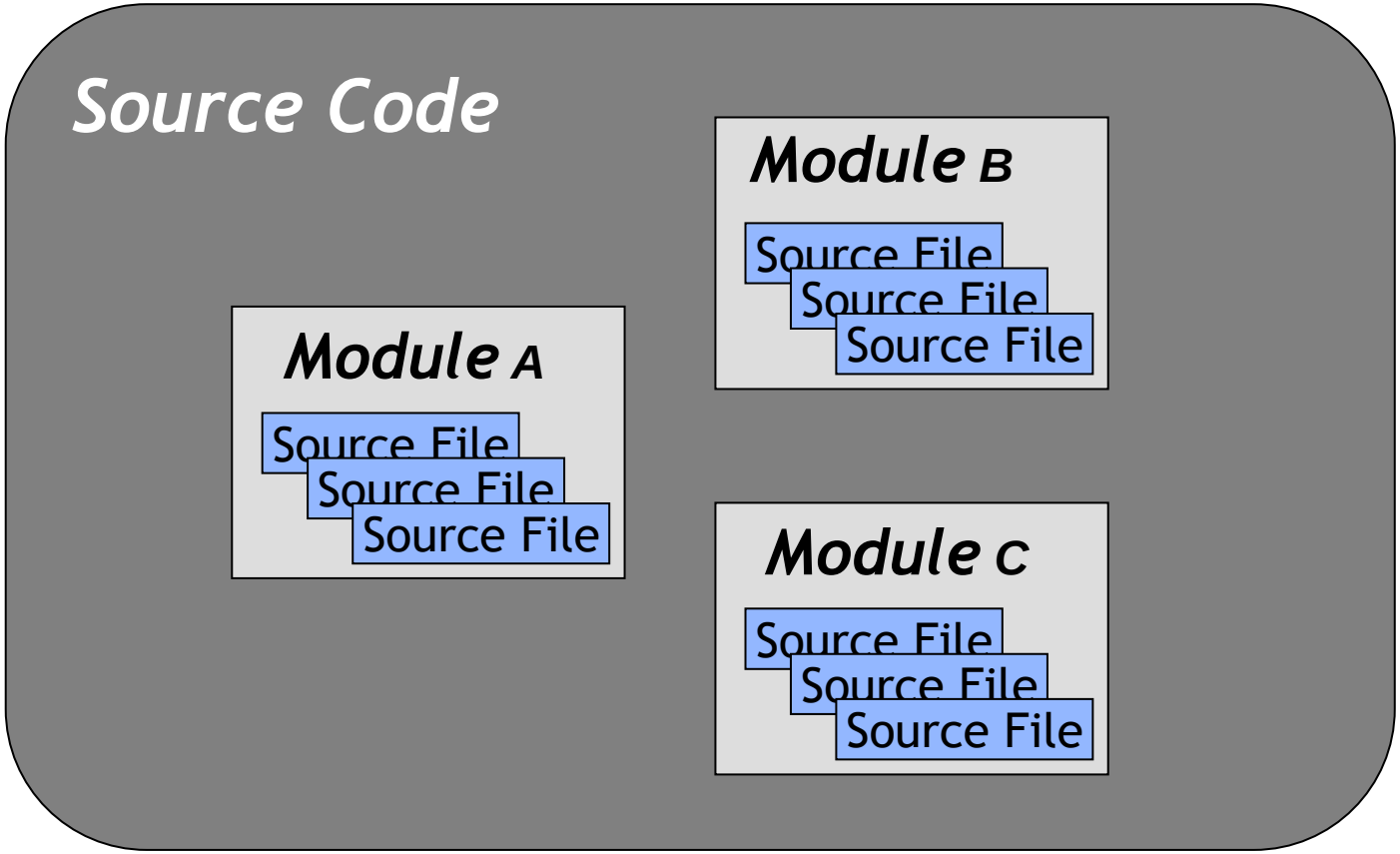
```
1  test1 []
2  ... lib1 [test1]
3  ... ... strings [lib1]
4          >> LIB: strings.a
5      > LIB: strings.a
6      >> LIB: lib1.a strings.a
7  >> LIB: lib1.a strings.a
```

Freemason

Architecture

RADVISION®
Delivering the Visual Experience™

The Freemason – How does It Work?



The Freemason – Targets

Products may be built for numerous targets using a selection of tools (i.e., compilers).

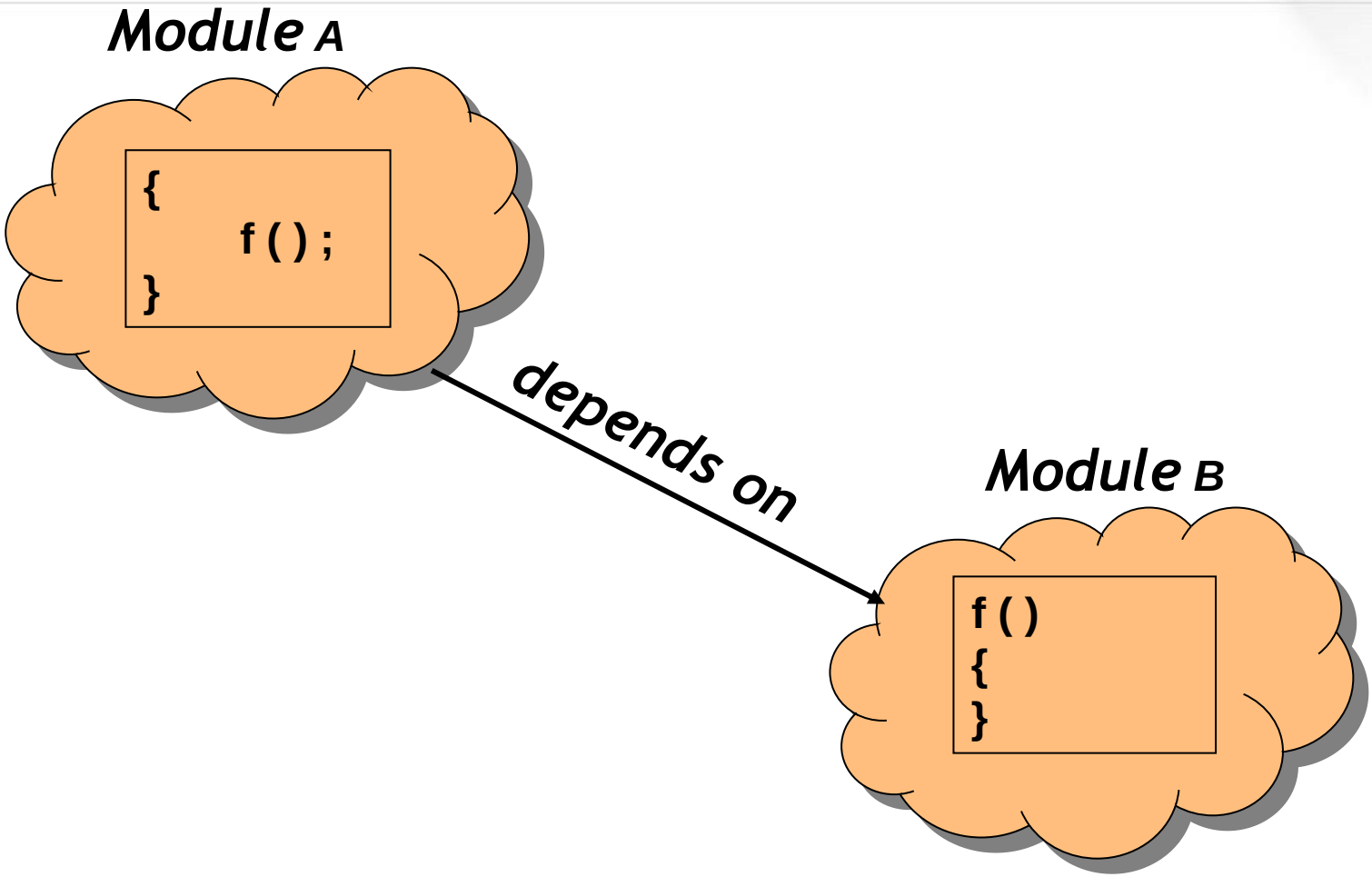
In order for Freemason to get a clear notion of what is expected from it, there is a needs to only specify the target platform.

The Freemason – Module Dependencies

- Apart from its source files, a module may require the products of other modules in order to build its own.
- If products of dependant modules are not available in advance, they should be built before they are required by their dependees.

Freemason allows modules to specify their dependencies and efficiently takes care of the recursive build operation in a process called ***deep build****.*

Module Dependencies



Freemason

Architecture

RADVISION®
Delivering the Visual Experience™

Freemason Architecture - Framework

- Freemason is essentially a combination of makefiles, organized in a hierarchical structure called Framework.
- The framework files that are used during a Freemason session depending on the session configuration (that is, values of configuration variables, such as PLATFORM or DEBUG).

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.

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).
- 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.

GNU Make

- The engine that drives Freemason is GNU make with syntax extension.
- 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.

Modules

Freemason module is either a white-box, a black-box, or both.

A black-box module:

- Able to build itself from its source files.
- Can be referenced and used by other modules (i.e., a module that represents a library that can be linked to an application).
- Owns a defs.mk file.

A White-box module:

- Owns a makefile file.

- Both kind of modules may have dependant modules.

Module Dependencies

Both White box and black box modules may have dependant modules.

- Module dependencies are described in a ***depends.mk*** files which 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.

Environment Variables

- In order to avoid mis-configuration of the build process, Freemason intentionally tries to avoid using of environment variables.
- All configuration aspects should be handled from within the Freemason module configuration files or from the Framework.

Freemason

Procedures

RADVISION®
Delivering the Visual Experience™

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.

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.

Interaction – Basic Examples (1)

Build a module for Windows platform with debug info:

```
mk PLATFORM=rv-win32 DEBUG=1
```

Build an optimized module for the RV755 platform:

```
mk PLATFORM=rv-755 OPT=1
```

Build an optimized module for TAMAR platform:

```
mk PLATFORM=rv-tamar OPT=1
```

Interaction – Basic Examples (2)

Build a module along with all its dependencies:

```
mk PLATFORM=... DEEP=1
```

Clean all build products for a given module:

```
mk PLATFORM=... clean
```

Modules

- A module owns three definition files, describing its properties:
  - ***makefile***
  - ***defs.mk***
  - ***depends.mk***
- TBD deep build
- TBD prebuilt products

Product Configuration

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

Target Operating System Configuration

Supported target operating systems include:

- win32 Windows
- vxworks-5.5 VxWorks 5.5
- vxworks-6.3 VxWorks 6.3

Target Platform Configuration

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

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.

The Repository

- Note: when traveling.

Modules Revisited

C/C++ Preprocessing Facilities

Freemason

Chapter Heading - TBD

RADVISION®
Delivering the Visual Experience™

Make Targets

- cc
- cpp
- clean
- install
- show-deps

Variables

- DEEP
- SHOW_CMD
- SHOW_DEPS
- SHOW_DIFFUSE
- FILE

Useful MAKE options

- -n
- -jN
- -k
- -B
- -P
- -v

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

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)
- *asn.1* ASN.1 compiler

More Development Tools

- The tools that 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)

Future Tool Support

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)

Freemason

Troubleshooting

RADVISION®
Delivering the Visual Experience™