1 Change History

Date	Version	Reason and Change
11/01/01	1.0.0	Initial release
10/07/01	1.0.1	Updated to reflect changes made to the make system.
12/07/01	1.0.2	Added description of `MKFLAGS`
17/07/01	1.0.3	Added description of `DVD_LINK_INIT`
28/09/01	1.0.4	Added `DVD_OS` info.
31/10/01	1.0.5	Added support for `UNIFIED_MEMORY` build and `SPECIAL_CONFIG_FILE`
09/05/02	1.0.6	Added support for generating a map file and suppressing the clean_all target.
09/09/02	1.0.7	Added support for SPARC build.
05/12/02	1.0.8	Added DVD_BUILD_VARIANT, OPTIONAL_CONFIG_FILE and OPTLEVEL information.
02/05/03	1.0.9	Added OS21 support information and update to the template for review.
12/05/03	1.0.10	Updates after review. Removed OS40 support.
29/08/03	1.0.11	Added mb390 to DVD_PLATFORM
09/10/03	1.0.12	Fixed OS21 support and added stpti4 to DVD_TRANSPORT options
15/06/04	1.0.13	Support mb391 and USE_DEBUG_KERNEL
29/06/04	1.0.14	Added PRESERVE_FILES flag
07/10/04	1.0.15	Support mb400.
09/02/05	1.016	Support Walkiry and mb411 boards.
29/03/05	1.0.17	Support mb390.

Date	Version	Reason and Change
17/11/2005	1.0.18	Support mb421, mb426, maly3s, mb395. Added demux to DVD_TRANSPORT. Added pc-cygwin to DVD_HOST. Updated examples for ST200.
20/01/2006	1.0.19	Support for mb428 (5525) & mb457 (5188).
18/04/2006	1.0.20	Support for mb436 (5107).
21/04/2006	1.0.21	Support for DTT5107 Refboard.
07/09/2006	1.0.22	Support for CAB5107 and SAT5107 Refboard.
22/02/2006	1.0.23	Support for mb519 (7200).
08/10/2007	1.0.24	Support for mb634 (5162).
13/02/2008	1.0.25	Support for mb618 (7111). Enabling 32 bit addressing support for ST40 devices. Overriding the default -mboard link option (OS21-ST40).
16/04/2008	1.0.26	Three environment options added: `STAPIREF_COMPAT`, `STAPIREF_INCLUDE_COMPAT`, `STPOWER_SUPPORTED`
17/06/2008	1.0.27	Support for mb680 (7105) and mb628(7141). Added environment option `DVD_BUILD_ONLY_CORE` to build only core module (and no ioctl) code under Linux.
23/06/2008	1.0.28	Added DVD_CPU environment variable to support multicore builds. eg: for STx7141.
08/07/2008	1.0.29	Support for mb671 (7200 cut2).
17/10/2008	1.0.30	Support for mb704 (5197).
25/08/2009	1.0.31	Added templates for LINUX supporting makefiles

2 Introduction

The STAPI make system provides the infrastructure for building the STAPI component libraries, their associated test suites and any sample applications. Its primary objective is therefore to build libraries and executables in both the development and installation environments. This document describes how to use the make system to build the appropriate libraries and how to add new libraries in the development environment.

This document describes version 2 of the STAPI make system. Version 1, is only considered in terms of its compatibility impact. Version 2 introduced many new features, such as object directories and provision for overriding many of the make settings. These changes were largely incompatible with the version 1, so a version 2 was introduced and the `MAKE_VERSION` variable used to distinguish between them.

3 Structure

Each STAPI component is designed such that it can be built in isolation or as part of a larger software component. The source tree structure has two variants, namely a development and installed tree.

The development tree, used internally within ST, is based within a number of clearcase VOBs (Versioned Object Bases). The root of this tree is typically `/dvd-vob` and each component has a its own VOB. For example, the following are some of the VOBs: `dvdca-prj-stevt`, `dvdgr-prj-stdenc` and `dvdbr-prj-stpti`. The structure of each VOB varies, but typically has `src`, `docs` and `tests` sub-directories. There are also specialized VOBs which hold general system header files, system configuration files and the make system files. The structure looks something like this:

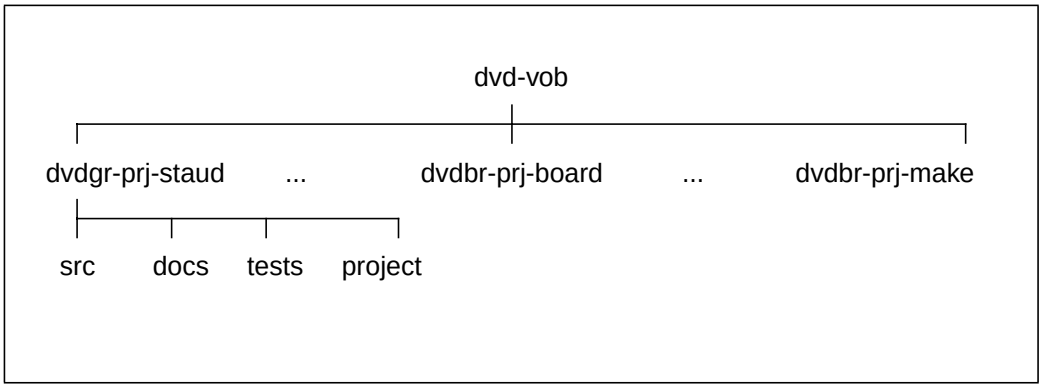

Figure 1 : A Development Tree

An installed tree has a very similar structure. The dvd-vob directory is replaced by a `src` directory and that is placed at the same level as an `include`, `docs`, `lib` and `config` directory. The component directories are renamed from the "`dvd`*XX*`-prj-`*COMPONENT*" format to "*COMPONENT*". Otherwise the component directories are much the same. The header files which are exported from various components are centralized in the top level include directory and static libraries are exported to the lib directory. The structure looks something like this:

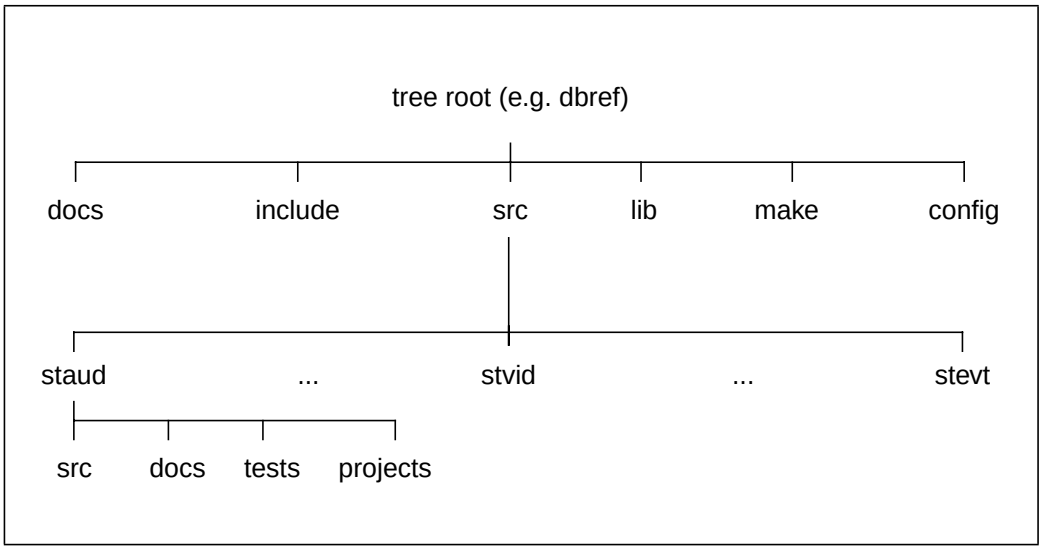

Figure 2 : An Installed Tree

It is worth noting that, relative to makesystem version 1, the general system header files have all moved to the `include` directory, the build configuration files have moved to the `config` directory and the make system files to the `make` directory (each set of files having been moved from specialized component VOBs).

4 Make System Basics

The make system was designed to allow a developer to set up some basic configuration options in the environment, making it possible to subsequently change directory into one of the STAPI component directories or its associated test directory and type “make”. The make system should then build whatever is appropriate in that directory. So, in the component directory, the make system will build the component library (e.g. `stdenc.lib` when building for an ST20 target in the STDENC component directory). In a component test directory, it should build one or more executables which could be downloaded and tested on a target platform.

To achieve either goal, the make system invariably needs to build the component plus any components on which it depends. Each component therefore defines a list of components that it requires (henceforth known as imports). Building the component therefore involves compiling all the parts in the component directory, followed by compiling all the parts in the imported components’ directories.

5 Structure of a Simple ST20 Makefile

Figure 3 lists the source for one of the simplest makefiles.

```
#    DVD_MAKE_VERSION := 2
*    ifdef IN_OBJECT_DIR

>    include $(DVD_MAKE)/generic.mak

     # Other components this component is dependant upon
1    IMPORTS := stpio
2    EXPORTS := sti2c.h sti2c.lib

3    TARGETS := sti2c.lib

     # local objects which comprise this component
4    OBJS := sti2c.tco

>    include $(DVD_MAKE)/defrules.mak

5    sti2c.lib: $(OBJS)
       $(BUILD_LIBRARY)

6    clean:
       @echo Cleaning sti2c
       -$(RM) $(OBJS)
       -$(RM) $(TARGETS)

     # Local dependencies
7    sti2c.tco: sti2c.h

*    else
*
*    include $(DVD_MAKE)/builddir.mak
*
*    endif
```

Figure 3 : A Simple ST20 Makefile

This makefile will build `sti2c.lib` for the ST20. The lines in the makefile can be separated into 4 categories:

1 The line marked with “#” indicates that this makefile is a new version makefile. This line will be ignored in further discussion, but all new makefiles and makefile updates must include this line.

2 Lines marked with “*” add object directory support to a makefile (see Section 5.3.)

3 Lines marked with “>” import the major portion of the make system files.

4 All other lines (numbered 1 to 7, above) provide the information pertinent to the building of the STi2C component.

All makefiles must have this basic structure and parts that fall into category 4 should be tailored for a particular component. The makefile can be generalized to the following format:

```
DVD_MAKE_VERSION := 2
ifdef IN_OBJECT_DIR

include $(DVD_MAKE)/generic.mak

Define the component-specific variables

include $(DVD_MAKE)/defrules.mak

List the rules to build component-specific targets

else

include $(DVD_MAKE)/builddir.mak

endif
```

Figure 4 : The Basic Makefile Structure

5.1 Component Variables

The initial set of variables required for a simple component makefile are:

1 IMPORTS

2 EXPORTS

3 TARGETS

The sample makefile introduced another variable (OBJS), but that is used internally by the makefile and not by the make system. It is, however, suggested that this model be followed because it makes for a readable makefile.

5.1.1 IMPORTS

This defines a list of STAPI components that are used by this component. So the STI2C component uses STPIO (line 1 of Figure 3).

5.1.2 EXPORTS

This defines a list of files which are exported by this component. Typically, a component will export a header file (which defines its interface) and a library. So in the example, STI2C exports `sti2c.h` and `sti2c.lib` (on an ST20 target). See line 2 of Figure 3.

5.1.3 TARGETS

This defines a list of files to be built for the component. This should minimally include any libraries listed in the EXPORT list, but may include any number of targets. In the example, `sti2c.lib` is to be built (see line 3 of Figure 3).

5.2 Component Rules

The component rules define the process used to build the exported libraries and any intermediary files. The example makefile defines the three targets which are required for the STI2C component:

1 The export target rules.

2 A “clean” target rule.

3 Header dependency rules.

5.2.1 Export Target Rules

The makefile must supply a rule for each of the targets listed in the TARGETS variable. In the example, the makefile supplied a rule to build `sti2c.lib` (see the line 5 of Figure 3). This rule defines that building `sti2c.lib` relies on `sti2c.tco`, and the BUILD_LIBRARY macro must be used to produce the library from the list of objects.

5.2.2 “clean” Target Rule

All makefiles must provide a “clean” target that defines how to remove any intermediary files. The sample makefile lists this target on line 6 of Figure 3. Typically, it defines a list of commands to delete the recognized intermediary files. These commands must be defines in terms of the RM macro because the makefile must work on both PC- and Unix-based systems.

5.2.3 Header Dependency Rules

This is an optional part of the component rules. This section usually defines which header files a particular object file depends on. To reduce the maintenance overhead, this is usually limited to a subset of STAPI header files (possibly only the ones that occur in the component directory), as can be seen in line 7 of Figure 3.

5.3 Object Directory Support

Lines marked with “*” in Figure 3 form part of the object directory support. This make system feature results in the creation of an object directory for a particular architecture, into which all intermediary files and libraries are stored. For an ST20 build, the object files will therefore be located in `objs/ST20`. All makefiles should be updated to include this support, because it allows components to be built for different architectures without cleaning the build tree. From release 2.7.0, you can override the default name of the object directory by setting the required name in DVD_BUILD_VARIANT.

5.4 Make System Support

The two lines marked with “>” in Figure 3 include the major part of the make system into a makefile. These files set up a number of variables and targets which allows the same make file to support various make system features from the same, simple makefile. It should be emphasized that the order of the makefile parts shown in Figure 3 and Figure 4 is important; failure to use this order can result in unexpected behavior.

5.5 Using the Makefile

Given the makefile in Figure 3, we need to know how to use it. The basic goal is to[1]:

1 Set some configuration options in the environment.

2 Invoke “make” with a target that we wish to build.

1. This assumes that the appropriate GNU make and compiler are installed on the build host. It is beyond the scope of this document to deal with the installation of these tools.

5.5.1 Setting Configuration Environment

The configuration environment is used to locate the make system files and components in within the directory structure of the build host. Minimally, the following three environment variables should be set:

1 `DVD_ROOT`

This should provide the location of the root of the STAPI components. In a development environment this would typically be `/dvd-vob`. In an installation environment it would be something like `/install_path/src` for a Solaris installation and `c:\install_path\src` for a DOS installation.

2 `DVD_MAKE`

This should provide the location of the STAPI make system files. In a development environment this would typically be `/dvd-vob/dvdbr-prj-make`. In an installation environment it would be something like `/install_path/make` for a Solaris installation and `c:\install_path\make` for a DOS installation.

3 `DVD_INCLUDE`

This provides the location of the central STAPI include directory. It is not necessary to set this in a development environment because the make system will use the header files from their location within the component directories (a process called in-place includes). In an installation environment, this would be set to something like `/install_path/include` for a Solaris installation and `c:\install_path\include` for a DOS installation.

There are a number of other configuration entries that can be set in the environment, but they will be described later.

5.5.2 Invoke “make”

This is the stage where the build process is invoked. To perform the default build, the command “make” is invoked (the actual command may be “gmake”, depending how the GNU make has been installed; for brevity we will just refer to the command as “make” in this document). This will build the `sti2c.lib` file. It is possible to invoke “make” differently to build the following targets:

- `make sti2c.tco`
- `make clean`

Note that it will be necessary to set up extra files like a `targets.cfg` file for the building and running of executables to succeed. It is beyond the scope of this document to deal with this part of the build process.

6 Doing More

The example in Section 5 is the simplest of makefiles. This section explains how to add to this basic makefile to deal with more complex requirements. Each sub-section deals with a particular goal. All sections deal specifically with ST20 makefiles, until multi-architecture makefiles are introduced in Section 6.13.

6.1 Building Imported Library Components

The STI2C makefile will only build the `sti2c.lib` and not the import libraries. There are two ways to get the make system to build the import libraries:

1 Invoke the build with a call "`make DVD_DEPENDS=all`".

This builds all imported libraries and for each of these libraries, all of their imported libraries are recursively built too, ... until all libraries are built. This ensures that all libraries required by a particular component are available. This is only useful within the development environment. See Section 7.16 for information about `DVD_DEPENDS`.

2 Change line 5 in Figure 3 to be:

```
sti2c.lib: $(OBJS) $(IMPORT_LIBS)
```

This instructs the make system to build the imported libraries before creating the `sti2c.lib`. This should generally be avoided for libraries, because the side-effect of this line is that the imported libraries will be added to `sti2c.lib`.

The `$(IMPORT_LIBS)` is usually added as a dependency when building an executable, e.g. for a test harness. The following makefile stub provides such an example:

```
...
IMPORTS := sttbx stuart stpio
TARGETS := myprog.lku

OBJS := one.tco two.tco three.tco

myprog_INIT := entry_point

include $(DVD_MAKE)/defrules.mak

myprog.lku: $(OBJS) $(IMPORT_LIBS)
   $(LINK_EXECUTABLE)

clean:
   @echo Cleaning myprog
   -$(RM) $(OBJS)
   -$(RM) $(TARGETS)
...
```

Figure 5 : Sample Makefile to Build An Executable

Note The line defining `myprog_INIT` is required by the `LINK_EXECUTABLE` macro to set the entry point to `myprog.lku`. See Section 10 for information about make system macros.

6.2 Adding Compile Flags

A common requirement is to update the compile flags in a makefile. It is possible to adjust the flags globally or for a single compile target.

6.2.1 Global Change

By adjusting the global `CFLAGS` variable, the makefile is able to change the build parameters for all objects compiled by that makefile. So, given the makefile in Figure 5, we could add the following line after the `TARGETS` line:

```
CFLAGS := $(CFLAGS) -DMY_DEFINE=1
```

Note the following important points regarding this additional line:

- The use of the ":=" assignment operator is important because it allows subsequent lines in the makefile to update the CFLAGS in the same manner. The "+=" assignment operator may also be used, but the "=" operator must not be used.
- A further important point is the location of the line within the makefile. The basic CFLAGS assignment occurs when the makefile imports the generic.mak file. Any subsequent adjustments of the CFLAGS can work on that basis.
- Include paths must not be added to the CFLAGS (see Section 6.3 for a description on how to do this).

6.2.2 Single Target Change

It is possible to define a new variable to set a compile flag for a single target. In Figure 5, if we wished to define "EXTRA_DEFINE=1" when building one.tco, all you need to do is add the following line before including defrules.mak:

```
one_CFLAGS := -DEXTRA_DEFINE=1
```

It is important not to add include paths using this variable, since another mechanism is provided for this purpose (see Section 6.3).

6.3 Adding to the Include Path

6.3.1 Global Addition

A common requirement is to add extra directories to the include path to locate required headers. The correct way to do that is to assign a value to the INCLUDE_PATH variable before including defrules.mak. As an example, the video driver splits its files over a few directories (currently: api, avsync, buffers, decode, diginput, display and trickmod). If compilation of a file required header files located in each of these directories, the following lines would achieve that:

```
MY_SUBDIRS := api avsync buffers decode diginput display trickmod
INCLUDE_PATH += $(addprefix $(DVD_BUILD_DIR)/,$(MY_SUBDIRS))
```

The result of the two lines would be to append a fully qualified path for each sub-directory to the include path. The path must be fully qualified for object directory builds. This is achieved by prepending "$(DVD_BUILD_DIR)/" to each of the sub-directories. DVD_BUILD_DIR is set by the make system to be the location of the build directory.

See Section 6.9.2 for information about sub-libraries.

6.3.2 Automatic Include Path

To understand when to add items to the include path, this section describes what the include path is initially set to.

When DVD_INCLUDE is set to indicate an include directory (with all generic, platform, architecture-dependent and STAPI header files), the path will include the following directories (in the indicated order):

1 Component build directory.

2 The include directory indicated by DVD_INCLUDE.

3 The full path to components listed in the makefile variable HEADER_IMPORTS.

4 The optional directory indicated by DVD_INCLUDE_EXPORTS.

5 Chip, board and platform directories.

When the DVD_INCLUDE is not set, the make system will use a process of in-place includes. As such the include path is much longer and includes the following:

1 Component build directory.

2 The full path to components listed in the makefile variable HEADER_IMPORTS.

3 The full path to components listed in the makefile variables IMPORTS or ST20_IMPORTS for ST20 targets or ST40_IMPORTS for ST40 targets.

4 The full path to the special include, chip, board and platform components which only store general header files (not STAPI header files).

6.4 Adding Link Flags

6.4.1 Global Additions

When linking an executable it is possible that a makefile may wish to add extra flags to the link command. Given the example in Figure 5 on page 9, it is possible to get the link process to produce a map file by adding the following line after the “TARGETS” line:

```
LKFLAGS = -M $(basename $@).map
```

A few important notes about this line:

- The makefile only works for the ST20 architecture, so the flags are therefore specific to the ST20 tools.
- The use of the “=” is important because it defers the evaluation of the “$@” until myprog.lku is linked (“$@” will become myprog.lku).
- The function basename strips the extension from myprog.lku to give myprog. This allows the makefile to produce a sensible map file called myprog.map.

6.4.2 Single Target Addition

The flags added in Section 6.4.1 will be applied to all targets linked in a single makefile (so if one.lku and two.lku are linked, both will have the additional link flags). It is possible to apply the flags to a single target by defining a line like this:

```
one_LKFLAGS := -M one.map
```

Given this line in a makefile which builds both one.lku and two.lku, the additional flag will only be applied when building one.lku. Either “:=” or “=” may be used in this assignment.

6.5 Adding to the Library Path

When a library is created or an executable is linked, the build tools will search a path to find the objects and libraries which go into the build. A makefile may therefore wish to extend the path to include directories in which extra libraries are located. For example:

```
LINK_PATH += $(DVD_BUILD_DIR)/extra_dir/objs/$(OBJECT_DIRECTORY)
```

Points to note about this addition:

- The path is a space separated list of directories to search.

- It is important to know where the object files are stored for any items added to the path; i.e. if a directory supports object directories, the path must refer to the appropriate object directory. The sample line assumes that the makefile in extra_dir supports object directories and appends "`/objs/$(OBJECT_DIRECTORY)`" to the directory name.
- It is unnecessary to add paths to imported libraries (those that appear in the `IMPORTS` list) or sub-libraries (those that appear in `SUBDIRS` list - see Section 6.9.2). These are automatically added by the make system, as necessary.
- This instruction only adds a location in which libraries are to be found. Adding a library requires the addition of a make dependency. This example adds `hardware.lib` as a dependency of `myapp.lku`, and that is located in the `hwlib` directory:

```
...
LINK_PATH += $(DVD_BUILD_DIR)/hwlib/objs/$(OBJECT_DIRECTORY)
...
myapp.lku: $(OBJS) $(IMPORT_LIBS) hardware.lib
   $(ECHO) Linking $@
   $(LINK_EXECUTABLE)
...
```

Figure 6 : Adding a Special Library and Path

6.5.1 Automatic Library Path

In order to know whether to add directories to a path, it is necessary to know what the path is initially set to.

When `DVD_EXPORTS` is defined, exported libraries are copied into the `DVD_EXPORTS` directory as they are built. For this build type, the library path will include the following directories (in the indicated order):

1 Any paths set in the `LINK_PATH` variable.

2 Sub-library paths (as specified in `SUBDIRS` variable) - see Section 6.9.2.

3 The `DVD_EXPORTS` directory.

4 A set of directories to search for *.cfg files. These include chip, board and platform config directories. It also includes `DVD_TARGET_PATH` directory, if specified.

For an in-place link (when `DVD_EXPORTS` is not defined), the path will include the following directories:

1 Any paths set in the `LINK_PATH` variable.

2 Sub-library paths (as specified in `SUBDIRS` variable) - see Section 6.9.2.

3 A path entry for the object directory of each imported component. So, if a makefile imports STPIO and STUART, the path will include fully-qualified directories to both object directories. The directory name may also include the appended "`/objs/$(OBJECT_DIRECTORY)`", depending if the particular component supports object directories.

4 A set of directories to search for *.cfg files. These include chip, board and platform config directories. It also includes `DVD_TARGET_PATH` directory, if specified.

6.6 Adding a Library Target

This section describes the steps involved in adding a library target to a makefile. Take care to add the respective variables and targets in the appropriate section in the makefile, as defined in Section 5 of this document.

1 Add the name of the library to the TARGETS line. (e.g. add `myapi.lib` to the TARGETS line).

2 If the library is to be exported (i.e. copied into the public libraries directory specified in the DVD_EXPORTS variable), add the library and its associated header file to the EXPORTS variable. It is a basic requirement that a header file must accompany the export of a library. If the library is local only (a sub-library which will later be incorporated into a higher-level library), it should not be added to the EXPORTS line - the EXPORTS line can be empty.

3 Add any components used by the library to the IMPORTS line (if they don't already exist). This excludes components like STSYS which are header-only imports (see next point).

4 Add any header-only imports (like STSYS or STBLIT) required by this library to the HEADER_IMPORTS line. For example the makefile could have the following line:

```
HEADER_IMPORTS := stsys stblit
```

5 Define a new variable (like OBJS) which takes a list of object files which make up the library. All sources (`*.c` files) for these objects should exist in the build directory. For example, the makefile may have the following line:

```
OBJS := one.tco two.tco three.tco
```

It is important to understand that the make system has pre-defines rules which understand how to produce `*.tco` from `*.c` (the "`*`" part will be named consistently). It is therefore imperative for the objects listed to be named appropriately.

6 Define a target which describes how to build the library:

```
myapi.lib: $(OBJS)
    $(BUILD_LIBRARY)
```

Take care to indent the `$(BUILD_LIBRARY)` macro with a TAB and not spaces, or this will cause an error in the build (this will result in a "missing separator" error in the makefile).

7 Make sure that the "clean" target deletes all intermediary files and targets introduced. For example, the "clean" target could look like this:

```
clean:
    @$(ECHO) Cleaning $(TARGETS)
    -$(RM) $(OBJS)
    -$(RM) $(TARGETS)
```

Ensure that the commands use the `$(ECHO)` and `$(RM)` variables as this ensures portability across PC and Solaris machines. Furthermore, do not add any command flags which will compromise this portablility. The addition of the "`-`" (minus) before the `$(RM)` commands will mean that the make will continue even if errors occur in the command.

6.7 Adding an Executable Target

This section deals with building an executable target within a makefile. Take care to add the respective variables and targets in the appropriate section in the makefile, as defined in Section 5 of this document.

1 Add the name of the executable to the TARGETS line (e.g. add myapp.lku)

2 Add any components to the IMPORTS line, if they do not already exist as an import. A rule-of-thumb is that any component header file used should be added as an import. This excludes components like STSYS (see next point).

3 Update the HEADER_IMPORTS line to include any header-only import components. This includes components like STSYS and STCOUNT.

4 Define a new variable that lists the object files which make up the executable. All sources (*.c files) for the build objects should exist in the build directory. The makefile may include the following line:

```
APP_OBJS := app1.tco app2.tco
```

See note for item 5 of Section 6.6 - it applies to this section too.

5 Define a new variable which sets the linker entry point for the executable. For our example so far, we may have the following line in the makefile:

```
myapp_INIT := board_init
```

Note that this is only necessary if DVD_LINK_INIT has not been set or if you wish to override the value of DVD_LINK_INIT (Section 7.15).

For the ST20, this value is used as the -p parameter to the link operation. For any further description of this, refer to the toolset documentation.

6 Optionally define a new variable which can add link flags to the link process. See Section 6.4.2 for further information.

7 Define a target which describes how to build the executable. The following line may be sufficient:

```
myapp.lku: $(APP_OBJS) $(IMPORT_LIBS)
    $(LINK_EXECUTABLE)
```

The executable myapp.lku will include the two object files listed in APP_OBJS, the libraries from the imported components and the ST20 libraries (the latter are automatically added by the LINK_EXECUTABLE macro).

When building the executable, the make system will traverse the component directories listed in the IMPORTS to ensure that their libraries are built. It is necessary for the IMPORTS line to include all components used (including those not directly referenced, but only used by imported components) otherwise not all libraries will be included in the link.

Take care to indent the $(LINK_EXECUTABLE) *macro with a TAB and not spaces, or this will cause an error in the build (this will result in a "missing separator" error in the makefile)*.

8 Make sure that the "clean" target deletes all intermediary files and targets introduced. For example, the "clean" target could look like this:

```
clean:
    @$(ECHO) Cleaning $(TARGETS)
    -$(RM) $(APP_OBJS)
    -$(RM) $(TARGETS)
```

Ensure that the commands use the $(ECHO) *and* $(RM) *variables as this ensures portability across PC and Solaris machines. Furthermore, do not add any command flags which will compromise this portablility. The addition of the "-" (minus) before the* $(RM) *commands will mean that the make will continue even if errors occur in the command.*

6.8 Uploading and Running an Executable

Uploading an executable to target hardware via a JEI or microconnect can be achieved through the make system. It is beyond the scope to describe this process fully, except that the make system includes a special target to automate the running of the executable.

If there is only one executable built by the makefile, it is easy to run the executable with the command:

```
make run TARGET=jei_name
```

If there are a number of executables built by the makefile (in the TARGETS line), it is a little more complicated:

- The command above will run the first executable in the TARGETS list.
- Each of the target executables can be run by a special form of the “make run” command. If the TARGETS lists one.lku and two.lku, the following two commands will run the respective executable:

```
make one.lku_RUN TARGET=jei_name
```

or

```
make two.lku_RUN TARGET=jei_name
```

Note that the run target is not support for earlier version makefiles, because they invariably defined their own run target. A version 2 makefile is defined as a makefile with DVD_MAKE_VERSION set to 2 (as shown in all the makefile templates).

6.9 Building Sub-Libraries

6.9.1 In the Same Directory

This should be used to create a library that is linked with an application. The library may also be exported. The following sample makefile builds testapp.lib and then links that library into testapp.lku.

```
...

HEADER_IMPORTS := stsys stcount
IMPORTS := staud stavmem testtool

TARGETS := testapp.lib testapp.lku

APP_OBJS := main.tco
LIB_OBJS := init.tco audio_tests.tco

include $(DVD_MAKE)/defrules.mak

testapp_INIT := board_init
testapp.lku: $(APP_OBJS) testapp.lib $(IMPORT_LIBS)
        $(LINK_EXECUTABLE)

testapp.lib : $(LIB_OBJS)
        $(BUILD_LIBRARY)

clean:
        $(ECHO) Cleaning testapp
        -$(RM) $(APP_OBJS)
        -$(RM) $(LIB_OBJS)
        -$(RM) $(TARGETS)

...
```

Figure 7 : Building a Sub-Library in the Same Directory

Note the following important points about Figure 7:

- There are two targets `testapp.lib` and `testapp.lku`. It is not strictly necessary to list testapp.lib in the TARGETS line, as long as it is listed as a dependency for `testapp.lku`. It is, however, good practice to list it in the TARGETS line to make it immediately obvious that a library is created in the build process.
- The library is not exported by this makefile (there is no EXPORTS line).
- The two variables APP_OBJS and LIB_OBJS contain a list of objects that form part of the executable and library respectively. These variables are used when defining the target for `testapp.lku` and `testapp.lib`.
- The separation of the `main.tco` from the library is generally done to separate the C entry function (`main()`) from the library (it is generally undesirable to include a function `main()` in a library). In this case the library is not exported, so the `main()` function could be included in the library.
- A sub-library in this case is actually unnecessary because of the small number of files that get inserted into the library and application.

6.9.2 In a Sub-Directory

Many STAPI components build sub-libraries which are combined to produce the component library. The following imaginary component makefile provides such an example.

```
1   ...
2
3   HEADER_IMPORTS := stsys
4   IMPORTS := stvid stvin stvout stlayer stpti stdenc
5   EXPORTS := sttla.lib sttla.h
6
7   TARGETS := sttla.lib
8   SUBDIRS := tlaapi enhance
9
10  OBJS := tla_init.tco sttla.tco tla_extra.tco
11
12  include $(DVD_MAKE)/defrules.mak
13
14  sttla.lib : $(OBJS) $(SUBDIR_LIBS)
15          $(BUILD_LIBRARY)
16
17  clean: $(SUBDIR_CLEAN)
18          $(ECHO) Cleaning $(TARGETS)
19          -$(RM) $(OBJS)
20          -$(RM) $(TARGETS)
21
22  ...
```

Figure 8 : Building a Sub-Library in a Subdirectory

Note the following important points about this example:

- The target for this directory is `sttla.lib`. The library includes three object files (`tla_init.tco`, `sttla.tco` and `tla_extra.tco`) and two sub-libraries (`tlaapi.lib` and `enhance.lib`). The sub-libraries are implicitly named based on the subdirectories that they are built in.
- Line 8 is important for the building of the sub-libraries. The list assigned to SUBDIRS is used by the make system to create a set of default build rules for the subdirectory libraries.
- The SUBDIR_LIBS listed as a dependency on line 14, ensures that the make system will build the sub-libraries when creating sttla.lib. This variable is automatically assigned by the make system.
- The libraries will automatically be added to `sttla.lib` by virtue of the inclusion of SUBDIR_LIBS as a dependency.
- The sub-libraries are built within the appropriate subdirectories. In order for the BUILD_LIBRARY macro to locate them, the subdirectories are automatically added to the library include path.
- The make system also automatically adds the subdirectories to the include path, so that the toplevel makefile can access any private header files in those directories.
- The final part of the puzzle is to ensure that the “`clean`” macro will clean the sub-library build too. This is achieved by adding the SUBDIR_CLEAN as a dependency to the “clean” target.
- The makefiles in the subdirectories don’t need any special requirements except that they produce a correctly named library.
- Quite often the makefiles in the subdirectories need to have access to header files in the parent directory. This can be achieved by adding the following line to the subdirectory makefile:

```
INCLUDE_PATH := $(dir $(DVD_BUILD_DIR)) $(INCLUDE_PATH)
```

6.10 Overriding Configuration (*.cfg) Files

When a user wishes to override configuration files, it is often desirable not to modify the files in the `config/board` directory (in an installed tree) or the board VOB (for a development tree). This can be achieved by setting an environment variable `DVD_USER_CONFIG` to refer to a private configuration directory. This directory should have a `platform`, `chip`, `board` and `block` sub-directories. Any configuration files placed in the board sub-directory will take precedence over files of the same name in the configuration directory or board VOB.

6.11 Adding Optional System CFLAGS in a Makefile

The make system creates a variable called `OPTIONAL_CFLAGS` which contains extra defines that may be appended to `CFLAGS`. This allows a makefile to easily pass configuration information to the compile process, when required. The following lines may be included in a makefile:

```
CFLAGS += $(OPTIONAL_CFLAGS)
```

or

```
audinit_CFLAGS += $(OPTIONAL_CFLAGS)
```

The following notes apply:

- The `OPTIONAL_CFLAGS` currently includes a define for the `DVD_TRANSPORT` variable of `-DDVD_TRANSPORT_`*`DVD_TRANSPORT`*. E.g: this may result in `DVD_TRANSPORT_PTI`.
- It also includes a define for each chip in the `CHIP_LIST` of `-DST_`*`CHIP`*. E.g. `-DST_7015`. See Section 6.12 for information about `CHIP_LIST`.
- See Section 6.2 for more information about setting `CFLAGS`.
- Other make system configuration files may add to the optional `CFLAGS`.
- C source files may test these defines as it would any other define.

6.12 Multi-Chip Support

This section deals with tasks associated with the multi-chip support, which allows the make system to automatically import the appropriate files to support a platform. Of primary interest is the types and location of the configuration files. There are 3 types of configuration files:

1 Platform configuration files (named *`platform`*`.mak`).

2 Chip configuration files (named *`chip`*`.mak`).

3 Block configuration files (named *`block`*`.mak`).

The location of these files differ between development and installation environments:

1 For development environments:

1.1 Platform configuration files are located in the platform VOB (`dvdbr-prj-platform`).

1.2 Chip configuration files are located in the chip VOB (`dvdbr-prj-chip`).

1.3 Block configuration files are located in the component VOB, depending which component they relate to (the file must exist in the same directory as the makefile which refers to it).

2 For installation environments:

2.1 Platform configuration files are located in the `platform` sub-directory of the `config` directory (this directory is at the same level as the `src`, `include` and `make` directories).

2.2 Chip configuration files are located in the `chip` sub-directory of the `config` directory.

2.3 Block configuration files are located in the component directory (the file must exist in the same directory as the makefile which refers to it).

6.12.1 Defining a New Platform File

This section describes how to create a new platform file in the platform VOB of the development environment. The following points should be considered when creating the platform file:

1 Decide on the name of the platform. The choice is based on one of two decisions:

1.1 If the platform is merely a variant of an existing platform (e.g. a variant of `mb282b`), it is unnecessary to create a new `DVD_PLATFORM` (see 1.2, below). Instead, take a name which indicates its relation to the original. This name would be assigned to `DVD_CONFIG_PLATFORM`. For example we may assign a name of `mb282b_myvariant`.

1.2 If the platform is not a variant, it would be necessary to create a complely new platform name which is assigned to `DVD_PLATFORM`. This addition will require a change to the make system files - the value of `DVD_FRONTEND` must be checked in `sysconf.mak`.

2 At this point either `DVD_PLATFORM` will refer to a new platform name or `DVD_CONFIG_PLATFORM` will refer to a new variant. The latter value will override any setting of the former for the purposes of multi-chip configuration. The overriding name will be used to load the equivalently named platform file from the platform VOB. For example, if the `DVD_CONFIG_PLATFORM` is set to `mb282b_myvariant`, the file `mb282b_myvariant.mak` file will be loaded from the platform VOB.

3 Create an appropriately named platform file in the platform VOB, based on 2, above. The platform file generally includes a variable `CHIP_LIST` which defines the list of chips found on that platform. E.g. :

```
CHIP_LIST := 5512 4600 stv0299 vg1011
```

3.1 The platform file can also provide platform configuration defaults. The variables that may be set are the following: `DVD_PLATFORM`, `DVD_FRONTEND`, `DVD_BACKEND`, `DVD_SERVICE` and `DVD_TRANSPORT`. The following lines may be found in a platform config file:

```
ifndef DVD_PLATFORM
    DVD_PLATFORM := mb282b
endif
ifndef DVD_BACKEND
    DVD_BACKEND := 7015
endif
ifndef DVD_TRANSPORT
    DVD_TRANSPORT := stpti
endif
```

3.2 The setting of `DVD_PLATFORM` in the platform configuration file, has an extra requirement: the platform configuration must be specified using the `DVD_CONFIG_PLATFORM` variable. This is to reduce confusion with `DVD_PLATFORM` changing value when the platform file is loaded.

4 For each of the chips listed in the `CHIP_LIST`, make sure that an appropriate chip file exists in the chip VOB (see Section 6.12.2).

6.12.2 Defining a New Chip File

This section describes how to create a new chip configuration file in the chip VOB of the development environment. The following points must be considered when creating the chip file:

1 Chip files are named according to the chip names listed in each CHIP_LIST line of platform configuration files. For each name in the list, a corresponding *chip*.mak must be created in the chip VOB.

2 An appropriately named chip configuration file should be created in the chip VOB. The file 5512.mak may look something like this:

```
BLOCKS := pti3 ... mpeg1cell
BLOCK_LIST := $(filter-out $(BLOCKS),$(BLOCK_LIST)) $(BLOCKS)
```

3 The first line defines the list of blocks used by this chip (this is a temporary variable). The ellipses used here indicate that there are likely to be more blocks listed (it is not some special makefile usage).

4 The second line updates the BLOCK_LIST with the blocks used by the chip - the line makes sure that the list has a unique list of blocks.

5 For each of the blocks in the BLOCK_LIST, an appropriate block configuration file must be created in the component directory where it is required (see Section 6.12.3).

6.12.3 Defining a New Block File

This section describes how to create a new block file in the component VOB of the development environment. The following points must be considered when creating a block file:

1 Block files are named according to the block names that appear in the BLOCK_LIST of a chip configuration file. These block files are located in the same directory as the component makefile which makes use of the block configuration. This means that this section should be defined in conjunction with the changes in a makefile listed in Section 6.12.4.

2 For each block requiring configuration in a particular makefile, create an *block*.mak file.

3 A block configuration file called mpeg1cell.mak may be create in the STAUD component directory and may look like this:

```
STAUD_MPEG1CELL := YES
```

4 The result of the included configuration files is that a makefile will be able to test for the existence of STAUD_MPEG1CELL to alter its build behavior. See Section 6.12.4 for a further discussion on how to achieve this.

6.12.4 Using Block Configuration in a Makefile

Given the definitions created by each of the block configuration files, a makefile needs to alter its build behavior. The following part of a makefile indicates how the STAUD makefile may take the definition of STAUD_MPEG1CELL into account:

```
...
OBJS := aud_api.tco aud_hal.tco ... aud_dbg.tco

ifdef STAUD_MPEG1CELL
   OBJS := $(OBJS) hal_mpeg1cell.tco
endif
...
```

Figure 9 : Adding Configured Object Files

Some important notes about this example:

- The inclusion of `hal_mpeg1cell.tco` is dependent on the definition of `STAUD_MPEG1CELL`. This only occurs when the `mpeg1cell.mak` is included.
- The use of ellipses is just to indicate missing parts of the makefile - this allows the example to concentrate on the important changes.
- The use of the "`:=`" assignment is important. This is known as an immediate assignment which allows the value of `OBJS` to be progressively updated. The "+=" immediate assignment may also be used. It is possible to use the "=" or recursive assignment in the correct circumstance. Since erroneous use of this will cause a makefile error it is better to use the "`:=`" and "+=" format only.

A makefile may also use the configuration to adjust the group of sub-libraries built for a platform. As an example, a makefile may add the following lines to a HAL library:

```
...
ifdef STAUD_MPEG1CELL
   SUBDIRS += mpeg1cell
endif
hal.lib: $(SUBDIR_LIBS)
   $(BUILD_LIBRARY)
...
```

Figure 10 : Adding Configured Sub-Libraries

6.12.5 Overriding Platform or Chip Configuration Files

The platform and chip configuration files are shipped as part of an installed tree in the directories indicated by Section 6.12. It is often a requirement to override the configuration files in a manner ensuring that changes will not be destroyed as part of a new release. This is especially true for customers. In a similar manner to the overriding the `*.cfg` files (as discussed in Section 6.10), platform, chip and block files may be overridden by files in the appropriate sub-directory of the `DVD_USER_CONFIG` directory (if set).

Platform files, defined in the same manner as described in Section 6.12.1, can be inserted in the `platform` sub-directory. These files will be used in preference to the system configuration files.

Chip files, defined in the same manner as described in Section 6.12.2, can be inserted in the `chip` sub-directory. These files will be used in preference to the system configuration files.

6.12.6 Adding Private Block Configuration Files

Referring to Section 6.12.3 and Section 6.12.5, new blocks referred to in the chip configuration files may be created privately (not in the component directory). These files can be inserted in the `block`

sub-directory of the DVD_USER_CONFIG directory. This directory is also searched for block configuration files and take precedence over component block files.

6.13 Adding Multi-Architecture Support

6.13.1 Introduction

A new feature in the make system provides support for ST40 and ST200 architecture builds. This is termed “multi-architecture support” within this document, as it allows makefiles to build objects for the ST20, ST40 and ST200. It is important to recognize that the use of object directories was introduced for the sole purpose of allowing binaries for multiple architectures to co-exist within the tree. As such, makefiles should support object directories in order to support ST40 and ST200 builds.

6.13.2 Modifying an ST20 Makefile to Support ST40

This section considers the changes required to convert a version 2, ST20 makefile to support ST40 and ST200 builds.

A library component makefile, like the one in Figure 3 on page 5, could be converted to look like this:

```
...
ST20_IMPORTS := stpio
ST20_EXPORTS := sti2c.h sti2c.lib
ST20_TARGETS := sti2c.lib

ST40_IMPORTS := $(ST20_IMPORTS)
ST40_EXPORTS := sti2c.h $(LIB_PREFIX)sti2c$(LIB_SUFFIX)
ST40_TARGETS := $(LIB_PREFIX)sti2c$(LIB_SUFFIX)

ST200_IMPORTS := $(ST20_IMPORTS)
ST200_EXPORTS := sti2c.h $(LIB_PREFIX)sti2c$(LIB_SUFFIX)
ST200_TARGETS := $(LIB_PREFIX)sti2c$(LIB_SUFFIX)

# local objects which comprise this component
ST20_OBJS := sti2c.tco
ST40_OBJS := $(call ST20OBJ_TO_ST40OBJ,$(ST20_OBJS))
ST200_OBJS := $(call ST20OBJ_TO_ST200OBJ,$(ST20_OBJS))

CFLAGS += -DBOTHCFLAGS
ST20_CFLAGS += -DMYST20FLAG
ST40_CFLAGS += -DMYST40FLAG
ST200_CFLAGS += -DMYST200FLAG

include $(DVD_MAKE)/defrules.mak

$(LIB_PREFIX)sti2c$(LIB_SUFFIX): $($(ARCHITECTURE)_OBJS)
   $(BUILD_LIBRARY)

clean:
   @echo Cleaning sti2c
   -$(RM) $($(ARCHITECTURE)_OBJS)
   -$(RM) $($(ARCHITECTURE)_TARGETS)
...
```

Figure 11 : A Combined ST20/ST40/ST200 Makefile

The following points apply to this figure:

- The ellipses indicate where the top and bottom of the makefile have been omitted for brevity.
- The make system refers to ST20_*XXX* variables when processing the makefile for an ST20 build. Similarly it uses ST40_*XXX* for a ST40 build and ST200_*XXX* for a ST200 build. For this reason, the basic objectives are to define the appropriate variables for the ST20, ST40 and ST200.
- The imports for this component are the same. As such, line 5 assigns the initial list to the ST40_IMPORTS and line 9 assigns to ST200_IMPORTS.
- The naming convention of objects, libraries and executables differs between the ST20 and ST40/ ST200. As such, lines like ST40_EXPORTS and ST40_TARGETS must name the targets appropriately. Since the naming convention for these objects is also defined, a macro can be called to automatically convert library and executable target names between the ST20 and ST40 and between ST20 and ST200. For example, the following lines could replace lines 6 and 7 for ST40 and lines 10 and 11 for ST200:

```
ST40_EXPORTS := $(call ST20LIB_TO_ST40LIB,$(ST20_EXPORTS))
ST40_TARGETS := $(call ST20LIB_TO_ST40LIB,$(ST20_TARGETS))

ST200_EXPORTS := $(call ST20LIB_TO_ST200LIB,$(ST20_EXPORTS))
ST200_TARGETS := $(call ST20LIB_TO_ST200LIB,$(ST20_TARGETS))
```

- The makefile defines three variables ST20_OBJS, ST40_OBJS and ST200_OBJS which list the objects in the ST20, ST40 and ST200 library respectively. The lines 15 and 16 could explicitly define the name of the object file:

```
ST40_OBJS := sti2c.o
ST200_OBJS := sti2c.o
```

- Lines 18 to 21 describe how compile flags are updated for ST20, ST40 and ST200 builds. Any flags assigned to CFLAGS will be applied for ST20, ST40 and ST200 builds. Flags assigned to ST20_CFLAGS, ST40_CFLAGS and ST200_CFLAGS will be applied for ST20, ST40 and ST200 builds respectively.
- Link flags are selectively applied to ST20, ST40 and ST200 in the same manner as indicated above. The applicable variables are LKFLAGS, ST20_LKFLAGS, ST40_LKFLAGS and ST200_LKFLAGS. This particular example does not involve a link phase, so link flags would be meaningless here.
- Include paths and link paths are unchanged for ST20, ST40 and ST200 builds.
- Lines 25 and 26 define a generalised rule for building the ST20, ST40 and ST200 export library. They have the appropriate object list variable as dependency and use the BUILD_LIBRARY macro. This macro is defined appropriately for an ST20, ST40 and ST200 build.
- The “clean” target (lines 28 to 31) is complex because the single target must work for ST20, ST40 and ST200. This relies on the similar naming of the ST20, ST40 and ST200 variables. For example, line 30 will be replaced by ST40_OBJS when building for an ST40 architecture and by ST40_OBJS when building for an ST200 architecture.

A makefile which includes executable targets, like the one listed in Figure 5 on page 9, could be converted to the following:

```
   ...
1  ST20_IMPORTS := sttbx stuart stpio
2  ST20_TARGETS := myprog.lku
3
4  ST40_IMPORTS := sttbx stpio
5  ST40_TARGETS := $(call ST20EXE_TO_ST40EXE,$(ST20_TARGETS))
6
7  ST200_IMPORTS := sttbx stpio
8  ST200_TARGETS := $(call ST20EXE_TO_ST200EXE,$(ST20_TARGETS))
9
10 ST20_OBJS := one.tco two.tco three.tco
11 ST40_OBJS := one.o two.o four.o
12 ST200_OBJS := one.o five.o
13
14 myprog_INIT := entry_point
15
16 include $(DVD_MAKE)/defrules.mak
17
18 myprog$(EXE_SUFFIX): $($(ARCHITECTURE)_OBJS) $(IMPORT_LIBS)
19    $(LINK_EXECUTABLE)
20
21 clean:
22    @echo Cleaning myprog
23    -$(RM) $($(ARCHITECTURE)_OBJS)
24    -$(RM) $($(ARCHITECTURE)_TARGETS)
   ...
```

Figure 12 : A Combined ST20/ST40/ST200 Makefile for Executables

The following points apply to this example:

- The ellipses indicate where the top and bottom of the makefile have been omitted for brevity.
- The make system refers to `ST20_`*XXX* variables when processing the makefile for an ST20 build. Similarly it uses `ST40_`*XXX* for an ST40 build and `ST200_`*XXX* `for a ST200 build`. For this reason, the basic objectives are to define the appropriate variables for the ST20, ST40 and ST200.
- The imports in this contrived example are different. As such, the `ST40_IMPORTS` and `ST200_IMPORTS` defines its own list of imports.
- Line 5 of the makefile shows how to automatically assign `ST40_TARGETS` while taking into account the differences in naming convention between the ST20 and ST40 architectures. Line 8 shows the same for a ST200 target.
- The makefile defines the variables `ST20_OBJS, ST40_OBJS` and `ST200_OBJS` which list the objects in the ST20, ST40 and ST200 executables. The ST40/ST200 executable includes different objects, so the new list is assigned to the variable.
- Lines 18 and 19 define a generalised rule for building the ST20, ST40 and ST200 executables. They have the appropriate object list variable as dependency.

  This rule includes `IMPORT_LIBS` as a dependency. This variable is defined appropriately for a ST20, ST40 and ST200 build.

  This rule uses the `LINK_EXECUTABLE` macro to build the target. This macro is also defined appropriately for the different architecture builds. The only difference between the ST20 and ST40/

ST200 use of this macro is that the ST20 version requires the definition of an entry point for the executable. This is defined on line 14.

- The “clean” target (lines 21 to 24) is complex because the single target must work for ST20, ST40 and ST200. This relies on the similar naming of the ST20, ST40 and ST200 variables. For example, line 23 will be replaced by `ST40_OBJS` when building for an ST40 architecture.
- The example in Figure 11 explains the method for customizing compile and link flags for multi-architecture builds.

6.13.3 Using Multi-Architecture Make

When a makefile supports ST20, ST40 and SPARC builds, the objects are built by one of the following commands:

- Build for ST20

```
make
```

or

```
make ARCHITECTURE=ST20
```

- Build for ST40

```
make ARCHITECTURE=ST40
```

- Build for SPARC

```
make ARCHITECTURE=SPARC
```

- Build for ST200

```
make ARCHITECTURE=ST200
```

The alternative to providing the architecture on the command-line is to set the entry in the environment. For example on a PC, the following will always build the ST40 architecture build:

```
set ARCHITECTURE=ST40
```

Note The *DVD_INCLUDE_EXPORTS* directory will be used as the location for exported header files, for ST20, ST40 and ST200 builds. These will have to be changed before running “make” if a different location is required.

6.14 Setting an OS21 Executable Region

Note This is only applicable to OS21.

An OS21 executable can be run in a particular region of memory or on the simulator. The make system allows the makefile to specify the region in the following way:

```
<exe_target>_REGION := p2
```

For example, overriding the placement region for target one.exe, the following line will suffice:

```
one_REGION := p2
```

If the region is not specified in the makefile, the value will default to the value of the `OS21_REGION` variable. This in turn, will default to a value of `p1`, if not set in the environment.

6.15 Setting the OS21 Runtime Library

Note This is only applicable to OS21.

By default, an OS21 executable is linked with the production library. This can be overridden to use the debug library by setting:

```
OS21_RUNTIME_LIB := os21_d
```

6.16 Passing Arguments When Running

The make system supports the passing of arguments when running executables for ST20, ST40 or ST200.

6.16.1 ST20

Set one of the following two variables when invoking make: `DVD_RUNARGS` or `ST20_RUNARGS`. For example:

```
make run TARGET=jei DVD_RUNARGS=args
```

or

```
make run TARGET=jei ST20_RUNARGS=args
```

Note:

- `DVD_RUNARGS` will be passed to ST20 and ST40 builds. `ST20_RUNARGS` will only be passed to ST20 builds.

6.16.2 ST40

Set one of the following two variables when invoking make: `DVD_RUNARGS` or `ST40_RUNARGS`. For example:

```
make run TARGET=jei DVD_RUNARGS=args
```

or

```
make run TARGET=jei ST40_RUNARGS=args
```

Note:

- `DVD_RUNARGS` will be passed to ST20 and ST40 builds. `ST40_RUNARGS` will only be passed to ST40 builds.

6.16.3 ST200

Set one of the following two variables when invoking make: `DVD_RUNARGS` or `ST200_RUNARGS`. For example:

```
make run TARGET=jei DVD_RUNARGS=args
```

or

```
make run TARGET=jei ST200_RUNARGS=args
```

Note:

- DVD_RUNARGS will be passed to ST20, ST40 and ST200 builds. ST200_RUNARGS will only be passed to ST200 builds.

6.17 SPARC Toolset Support

The make system also supports build operations using the SPARC toolset (DVD_TOOLSET set to SPARC). This is analogous to the ST40 support in that the following make system variables are defined in the makefile to build SPARC targets:

```
SPARC_TARGETS
SPARC_EXPORTS
SPARC_IMPORTS
```

6.18 Creating a New “Version 2” Makefile

This section describes the procedure for creating a new “version 2” makefile.

The makefile creation process will vary according to the targets to be built by the makefile. The following list describes the actions to be taken or points that need to be considered:

1 The first action is to acquire a template makefile which closely matches the requirements of the makefile. Refer to Appendix : *Makefile Templates* on page 46. This will provide the starting point for most makefiles.

2 Modify the makefile template as directed in the appendix - this will provide the rudiments for a makefile.

3 The makefile should now be modified with any optional parts. These may include the following parts:

3.1 Compile flags (see Section 6.2).

3.2 Include path (see Section 6.3).

3.3 Link flags (see Section 6.4).

3.4 Library path (see Section 6.5).

3.5 Sub-library builds (see Section 6.9).

3.6 Conditional parts based on configuration (see Section 6.12.4).

6.19 Converting an Existing Makefile to “Version 2”

The preferable approach when converting a makefile to “version 2” is to replace the makefile using the methods listed above. This will reduce any future maintenance issues.

6.20 LINUX OS Support

The make system also supports build operations for LINUX OS on ST40 platforms. The following make system variables are defined in the makefile to build on STLINUX

```
DVD_OS should be linux
KDIR should be defined as the path of the kernel.
```

7 Build Options

This section provides information on the various variables that can be set to modify the build behavior. These options are listed in the following sub-sections which describe their purpose.

7.1 Basic Options

These three option variables are usually set for a build (see Section 5.5.1 for more information):

- DVD_MAKE
- DVD_ROOT
- DVD_INCLUDE

When the DVD_INCLUDE option is not specified, the make system will try and perform in-place includes (i.e. it will try and use the component header files from their location in the VOB). This will not work in an installation environment. It is necessary for each component to ensure that all components that it uses are added to the IMPORTS or HEADER_IMPORTS part of the makefile. If the list is incomplete, the build of a component will not work.

When building a mixture of version 1 and version 2 makefiles, it may sometimes be necessary to define a variable "DVD_BASE_HEADER_IMPORTS". This is a list of components directories to add to the include path for version 1 makefiles. This is required to support in-place includes for version 1 makefiles. For example:

```
DVD_BASE_HEADER_IMPORTS := stevt stsys stcount
```

7.2 Exporting STAPI Libraries

Variables: EXPORTS, ST20_EXPORTS, ST40_EXPORTS, ST200_EXPORTS

- For ST20 builds, the make system will use ST20_EXPORTS or EXPORTS (if ST20_EXPORTS is not defined).
- For ST40 builds, the make system will use ST40_EXPORTS or EXPORTS (if ST40_EXPORTS is not defined).
- For SPARC builds, the make system will use SPARC_EXPORTS or EXPORTS (if SPARC_EXPORTS is not defined).
- For ST200 builds, the make system will use ST200_EXPORTS or EXPORTS (if ST200_EXPORTS is not defined).
- The variable must refer to a directory to which the STAPI libraries are to be copied when they are built.
- The directory must be writable by the user.
- If the variable is set, but the directory does not exist, it will be created.
- If the variable does not exist then the libraries will not be exported.
- If the variable does not exist, nor will a central repository for the built libraries. In this case the make system will attempt to use the libraries from their location within the tree (in both development and installation environments). If a components IMPORTS list is incomplete, this method will not work.

7.3 Exporting STAPI Headers

Variable: DVD_INCLUDE_EXPORTS

- The variable must refer to a directory to which the STAPI headers are to be copied during the build process.
- The directory must be writable by the user.
- If the variable does not exist, then nothing will be exported.
- If the variable exists but the directory does not, the directory will be created.

7.4 Future of DVD_FRONTEND and DVD_BACKEND

- These define the frontend (or main processor) and the backend (or video decode processor).
- These variables are effectively obsolete. The multi-chip support described in Section 6.12 allows more accurate configuration of platform-specific options.
- If not specifically set, the DVD_PLATFORM determines the values for DVD_FRONTEND and DVD_BACKEND.
- Makefiles may use the DVD_FRONTEND (e.g. 5510, 5512, TP3) and DVD_BACKEND to modify its behavior.
- Both values may be specified in a platform configuration file (see Section 6.12).

7.5 Building for ST20, ST40 and ST200

Variable: ARCHITECTURE

Options: ST20, ST40, STLINUX, ST200 or SPARC

Default: ST20

- This will instruct the make system to build for the ST20, ST40 or ST200. The appropriate build tools must be installed.

The specified architecture is used to set a CFLAG which is provided to all compilations. The CFLAG is -DARCHITECTURE_*ARCHITECTURE*. A C program can therefore test for the definition of ARCHITECTURE_ST20, ARCHITECTURE_ST40, ARCHITECTURE_LINUX , ARCHITECTURE_ST200 or ARCHITECTURE_SPARC.

7.6 Overriding Config Files

Variable: DVD_USER_CONFIG

- This allows the user to override the *.cfg files and multi-chip configuration files (see Section 6.10 and Section 6.12.5).
- The directory referred to in the variable must have a platform, board, chip and block sub-directory. Files must be placed in the appropriate sub-directory, according to the directions in the above references.

7.7 Specifying an Alternate Main Config File

Variable: SPECIAL_CONFIG_FILE

- The main config file is the “entry point” config file used when linking or running an executable. This generally depends on the setting of the DVD_PLATFORM. For example, if DVD_PLATFORM were set to mb282b, the “entry point” config file would be mb282b.cfg.
- Defining this variable in the build environment or in the platform config file will override any system default config file. For example, in the above example where DVD_PLATFORM is set to mb282b, setting SPECIAL_CONFIG_FILE to my_mb282b.cfg will mean that a file by that name will be used instead.
- The only requirement is that the config file exist on the search path, including the DVD_TARGET_PATH directory (see Section 7.6 and Section 7.10).

7.8 Specifying an Optional Config File

Variable: OPTIONAL_CONFIG_FILE

- The main config file is the “entry point” config file used when linking or running an executable. This generally depends on the setting of the DVD_PLATFORM. For example, if DVD_PLATFORM were set to mb282b, the “entry point” config file would be mb282b.cfg.
- Defining this variable in the build environment or will specify and optional override to the system default config file. For example, where DVD_PLATFORM is set to mb361, setting OPTIONAL_CONFIG_FILE to my_mb361.cfg will mean that a file by that name will be used instead (if it exists). If it does not exist, mb361.cfg will be used instead.
- The only requirement is that the config file exist on the search path, including the DVD_TARGET_PATH directory (see Section 7.6 and Section 7.10).

7.9 Specifying the Service

Variable: DVD_SERVICE

Options: DVB or DIRECTV

Default: DVB

- This variable allows makefiles to select optional parts depending on the required service.
- The DVD_SERVICE variable is used to set a CFLAG which is provided to all compilations. The CFLAG is -DST_*DVD_SERVICE*. A C program can therefor test for the definition of ST_DVB or ST_DIRECTV.
- This option can be set in the platform configuration file (see Section 6.12).

7.10 Path to targets.cfg

Variable: DVD_TARGET_PATH

Default: .

- This variable allows a user to specify the location of the targets.cfg file.
- It is recommended that the targets.cfg only exist in this directory, as it is one of the last locations on the search path.
- This allows the user to have a targets.cfg outside an installed tree - one that will not be modified when patches are applied.

7.11 Setting the Build Platform

Variable: DVD_PLATFORM

Options: ST20: mb231, mb282, mb282b, mb275, mb193, mb314, mb5518, mediaref, mb361, mb382, mb376, espresso, mb390, mb391, mb400, mb385, walkiry, maly3s, mb395, mb457, mb436, DTT5107, CAB5107, SAT5107, mb634

ST40: mb317a, mb317b,overdrive, mediaref, mb376, espresso, mb411, mb519, mb618, mb628, mb680, mb671, mb704

SPARC: explorer4010, explorer8010

ST200: mb390, mb421, mb426, mb428

Default: mb231

- This platform variable determines the DVD_FRONTEND, if it is not specifically set.
- The DVD_FRONTEND is then used to set the DVD_BACKEND if that is not specifically set.
- This option may be set in the platform configuration file (see Section 6.12).

7.12 Setting the Configure Platform

Variable: DVD_CONFIG_PLATFORM

Default: DVD_PLATFORM

- This variable is used to change the platform name that is used in the multi-chip support of the make system (see Section 6.12).
- When not set it will default to DVD_PLATFORM (see Section 7.11).

7.13 Setting the Build OS

Variable: DVD_OS

Options: OS20, OS21

Default: Based on ARCHITECTURE

- This variable is made available to the make system to vary build rules according to the operating system that the STAPI component is being built for.
- If not set, the value depends on the setting of ARCHITECTURE - OS20 for ST20 and OS21 for ST40/ST200.

Note From version 2.3.0 of the make system release, the default value has been changed to OS21 (from OS40), since OS40 is now defunct.

7.14 Setting the Build Host

Variable: DVD_HOST

Options: unix, pc, pc-cygwin, win98

Default: Based on detection of host type (unix/pc)

- This selects the type of host used for building.
- Setting of this is generally not required. Only required when wanting to build for Microsoft Windows 98, where “win98” is selected

Note Support for Microsoft Windows 98 is only partial, i.e. due to constraints placed by the Microsoft Windows 98 DOS box and the way that the make system deals with certain operations, operations like `make clean` do not work.

7.15 Setting the Linker Procedure

Variable: `DVD_LINK_INIT`

Default: `board_init`

- This variable is used to change the linker procedure invoked by the toolset during the link phase.
- When not set it will default to `board_init`).
- This option may be overridden in the target makefile by setting `<target>_INIT`.

7.16 Building Dependencies

Variable: `DVD_DEPENDS`

Options: `no`, `yes`, `top`, `all`

Default: `yes`

- This variable modifies the behavior of building `IMPORT_LIBS` as a dependency.
- The default behavior (yes), is to build all imported libraries if the variable `IMPORT_LIBS` is listed as a dependency of an object.
- Option “no” will suppress building of imported libraries that are listed as dependencies of an object. This should only be user if all required libraries have been built and not changed. It speeds up compilation by not traversing the library directories.
- Option “top” is used when building within a library directory to ensure that all imported libraries can be built. When compared to the “all” option, this option will only effect the build in the directory that “make” is first run.
- Option “all” will build all imported libraries for all components at all levels. This will invariably mean that the build will take a long time as it will traverse the same directories many times, when checking that all dependencies are satisfied.
- The variable could be set in the environment, but care should be taken to make sure that all dependencies are built when required.
- The most common use will be to run “make” as follows:

```
make DVD_DEPENDS=no
```

7.17 Changing Toolsets

Variable: `DVD_TOOLSET`

Options: `ST20`, `ST40, ST200, SPARC.`

Default: *ARCHITECTURE*

- This variable is offered for future expansion.
- It is provided for the situation where different compilers are used to build the STAPI libraries. For example, if the GCC compiler were used, the DVD_TOOLSET may be set to “gnu”. The make system would then be modified to support the compiler, allowing a different toolset to be used in the compilation..

7.18 Setting the Make Limit

Variable: DVD_MAKELIMIT

Default: 15

- The make system places a limit on the depth of recursive calls to “make” that it will allow. This variable sets that limit.
- Bear in mind that builds within an object directory “consume” two levels of recursion in the make tree.

7.19 Setting the Transport

Variable: DVD_TRANSPORT

Options: pti, link, stpti, stpti4 or demux

Default: for 8010 the default is demux, for others it is pti

- This selects the transport for a platform build.
- This variable is used to update the OPTIONAL_CFLAGS variable with the following define:

  -DST_*DVD_TRANSPORT*

7.20 Doing a Debug Build

Variable: DEBUG

- When building with DEBUG=1, the toolset definition will ensure that the compiler creates debug object files which will be linked into a debug executable.

7.21 Changing the compilation optimization

Variable: OPTLEVEL

Options: (toolset dependent) 0..3 for ST200 (0..2 for ST20)

- This will override the default optimization for a compilation. Setting this variable to 0 while doing a standard build will mean that all compiler optimization will be turned off.

7.22 Building a Unified Memory Object

Variable: UNIFIED_MEMORY

- Building with UNIFIED_MEMORY=1 will create an object file that runs in unified memory.
- Not all platforms or components will support a unified memory build.

- This must also be set when running an executable.
- This option may be set in the platform configuration file (see Section 6.12).

7.23 Creating a Specialized Build Variant

Variable: `DVD_BUILD_VARIANT`

- This allows the user to override the object directory used to store the compiled files (for makefiles which support object directory builds).
- Setting this to `stpti-dvb` will mean that the object directory will be called `objs/stpti-dvb` rather than `objs/ST20 or objs/ST40` or `objs/ST200`.
- From release 2.7.0, creation of subdirectory within object directory is supported. It is possible to create `objs/ST20/xyz.`

7.24 Building for Codetest

Variable: `DVD_CODETEST`

Options: `TRUE` or `FALSE` (undefined)

Default: `FALSE`

- Setting this to `TRUE` will get the make system to build an application with codetest support.
- If this is not defined or set to anything other than `TRUE`, codetest support will be omitted.
- It is beyond the scope of this document to provide more information about using codetest.

7.25 Generating a map file

Variable: `GENERATE_MAP`

- Building with `GENERATE_MAP=1` will generate a map file when linking an executable in a makefile which does not support map file generation.
- The make system automatically includes the toolset commands to generate a map file, so if the makefile already deals with this, do not define this variable. Doing so may cause the link to fail.

7.26 Use OS20 debug Kernel

Variable: `USE_DEBUG_KERNEL`

- linking with `USE_DEBUG_KERNEL=1` in the environment will make the st20 linker use the debug versions of the kernel with extra assertions and checking.

7.27 Suppressing the clean_all target

Variable: `SUPPRESS_CLEAN_ALL`

- Defining this variable at the top of a makefile causes the make system to suppress the default `clean_all` target.
- This is desirable in situation where the standard `clean_all` target does not perform the required operations or has some undesired side-effect (such as cleaning the exports directory).

7.28 Protecting files in object directories

Variable: PRESERVE_FILES

- Defining this will protect files not cleaned by a component makefile from being deleted. For example, running a test with -log output.log to capture the output to the DCU will create a file called output.log in the objects directory (objs/ST20). Running “make clean” without this flag defined in the environment will cause the output.log file to be deleted with the object directory. If this is defined, the make system will not forcibly remove the object directory and the file, thus preserving the log file (unless the makefile specifically deletes the file).

7.29 Performing Warning Checks using GCC

Variables: `GCC_CHECK`, `GCC_CHECK_SA` and `GCC_C99`

Options: defined or undefined

Default: undefined

- When invoking make as “`make GCC_CHECK=1`” or “`make GCC_CHECK_SA=1`”, the make system will invoke GCC to provide additional warning checks on the component source files. It does this by compiling the source files with warning checking on.
- Remember to clean the appropriate source modules before and after invoking make in this manner.
- A PC or Solaris build will require the prior installation of GCC for this option to work.
- `GCC_C99` can be set additionally to perform error/warning checks using the C99 standards as a reference.

7.30 Creating Object Dependencies

Variable: `GCC_DEP`

Options: defined or undefined

Default: undefined

- Invoking make as “`make GCC_DEP=1`” will get the make system to produce an object dependency file.
- This option should be invoked after cleaning the component so that a full list of dependencies are generated.
- The dependency files are named “`<source file>.d`”, for example “`vin_init.d`”. These will be created in the directory where object files are placed. For example `src/objs/ST20` of a component.
- The data should be extracted from these files and added to the `makefile` (excluding the absolute paths). After which, the dependency files should be removed as they are not cleaned by the make system.

7.31 Performing LINT Analysis

Variable: `LINT_OUTPUT`

Options: (provide a full output path to a file which will store the lint output information)

Example: `/usr/home/lintoutput.txt`

- If this is not defined, lint processing will not happen.
- Set this variable to the full path of a file to store the lint processing output. All files processed will be appended to this file.
- If a relative file is used, this output will be distributed within the tree and may cause a problem in cleaning as it may be placed in the object directory and not cleaned.
- On a PC, the LINT_PATH should also be set to indicate where the lint-nt.exe is placed.
- On Solaris, flexi-lint is used. The command used is “flint” and this must be on the execution path.
- Be careful not to compile too many component libraries with this option enabled as it will create a huge output file which will be unweildy.

The lint tool will only be invoked for files which require building, i.e. if the compiler will be invoked to create the object file, the lint tool will be invoked first. This provides a method to selectively analyze C source files. Conversely, it is important to clean a portion of the tree for which information is to be gathered to make sure that all the files are analyzed.

7.32 Enabling 32 bit addressing support for supported ST40 devices

Variable: `DVD_ADDRESSMODE`

Options: (32, 29, undefined)

Default: undefined

- Set as 29 or leave it undefined for building in the default 29 bit addressing mode for devices such as 71xx.
- Set as 32 for enabling and building with 32 bit addressing support.

7.33 Overriding the default -mboard link option (OS21-ST40)

Variable: `MAPPED_MBOARD`

Options: (provide the name of the required -mboard linktime procedure)

Example: `MAPPED_MBOARD=board_mb618_lmi0_0x05`

- Set the name of the required -mboard linktime procedure. The `OS21_REGION` value automatically gets suffixed.

The OS21_REGION value is by default 'se' for 32 bits mode and 'p1' for 29 bits mode which can be overrided by setting OS21_REGION in the build environment.

7.34 Power Management support and STPOWER

Variable: `STPOWER_SUPPORTED`

Options: (1, unset)

Example: `STPOWER_SUPPORTED=1`

- Set as 1, to enable power management support using STPOWER in drivers.

7.35 Using STAPIREF compatible code

Variable: `STAPIREF_COMPAT, STAPIREF_INCLUDE_COMPAT`

Options: (1, unset)

Example: `STAPIREF_COMPAT=1`

or

`STAPIREF_INCLUDE_COMPAT=1`

- Set `STAPIREF_COMPAT` as 1 to use stapiref compatible code. When unset stfae specific code is used.
- Set `STAPIREF_INCLUDE_COMPAT` as 1 typically in driver environments to check stfae code compilation when not having the same platform directory structure as in the stfae/sdk tree. Setting this option will use the stapiref specific board & chip header files keeping rest of the code stfae specific.
- The typical use cases for using these variables are:
  - Stfae doesn't set any of these two environment options to use stfae code.
  - Sdk tree doesn't set any of these two options to use stfae code.
  - Sdk tree sets STAPIREF_COMPAT=1 option to use stapiref code.
  - Driver owners set STAPIREF_COMPAT=1 to use stapiref code.

Driver owners set STAPIREF_INCLUDE_COMPAT=1 to compile stfae code.

7.36 Building for Multicores/Multi Host SOCs (eg: STx7141)

Variable: `DVD_CPU`

Options: (ESTB, ECM, unset)

Example: `DVD_CPU=ECM`

- Set as `ECM` to build for ECM core. Set as `ESTB` to build for ESTB core. When unset, defaults to `ESTB` for STx7141.
- Setting this variable, causes the `ST_$(DVD_CPU)` variable to be passed in CFLAGS during compilation. eg: For STx7141, either of `ST_ESTB` or `ST_ECM` will be passed in the CFLAGS.

8 Make System Variables

This section does not describe all the variables used in the make system, but those of importance to makefiles.

CFG_PATH	A space separated list of directories to search for *.cfg files. These config files are required during an executable link.
CFLAGS	Compiler flags used for both ST20 and ST40 compilations. Care must be taken with this to ensure that flags can be used in both architectures and for the selected toolset.
EXPORTS	Another name for ST20_EXPORTS.
HEADER_IMPORTS	A space-separated list of components that are required for their header files only (no objects are built in that directory). For example, STSYS may be added to this list.
IMPORT_LIBS	This variable is a list of libraries imported. It is created by expanding ST20_IMPORTS or ST40_IMPORTS (depending on the architecture selected) with the appropriate library naming convention. An example may be ("staud.lib stvid.lib stevt.lib" for an ST20).
IMPORTS	Another name for ST20_IMPORTS.
INCLUDE_PATH	A space-separated variable that can be modified to add directories to the include path. Remember to specify full paths to the include directory.
LINK_PATH	A space-separated list of directories to be added to the library include path.
LKFLAGS	Link flags used when linking both ST20 and ST40 executables. Care must be taken when using this to ensure that the flags can be used when linking executables for both architectures.
OPTIONAL_CFLAGS	This variable takes extra CFLAGS (usually C defines) which a makefile can add to the CFLAGS, if required.
SPARC_CFLAGS	Similar to ST20_CFLAGS for SPARC build.
SPARC_EXPORTS	Similar to ST20_EXPORTS for SPARC build.
SPARC_IMPORTS	Similar to ST20_IMPORTS for SPARC build.
SPARC_LKFLAGS	Similar to ST20_LKFLAGS for SPARC build.
SPARC_TARGETS	Similar to ST20_TARGETS for SPARC build.
ST20_CFLAGS	Compiler flags used for ST20 compilations.
ST20_EXPORTS	A space-separated list of header files and ST20 libraries exported by a component. Usually it will be something like "staud.h staud.lib".

ST20_IMPORTS	A space-separated list of components required to build for the ST20. Even if the component is a library, a full set of imports should be listed to support in-place includes and in-place linking (see Section 7.1 and Section 7.2).
ST20_LKFLAGS	Link flags used when linking ST20 executables.
ST20_TARGETS	A space-separated list of libraries and executables to be built for the ST20.
ST40_CFLAGS	Compiler flags used for ST40 compilations.
ST40_EXPORTS	Same as ST20_EXPORTS except that it list exports for an ST40 build.
ST40_IMPORTS	Same as ST20_IMPORTS except that it lists imports for an ST40 build.
ST40_LKFLAGS	Link flags used when linking ST40 executables.
ST40_TARGETS	Same as ST20_TARGETS except that it lists targets for an ST40 build.
ST200_CFLAGS	Compiler flags used for ST200 compilations.
ST200_EXPORTS	Same as ST20_EXPORTS except that it list exports for an ST200 build.
ST200_IMPORTS	Same as ST20_IMPORTS except that it lists imports for an ST200 build.
ST200_LKFLAGS	Link flags used when linking ST200 executables.
ST200_TARGETS	Same as ST20_TARGETS except that it lists targets for an ST200 build.
SUBDIRS	This is a space separated list of subdirectories to traverse to build the named sub-libraries.
SUPPRESS_CLEAN_ALL	Defining this at the top of a makefile causes the make system to suppress the clean_all target (see Section 7.27: *Suppressing the clean_all target* on page 34).
TARGETS	Another name for ST20_TARGETS.

9 Make System Targets

This section provides reference for targets defined by the make system. This does not list all targets, but only those of use to component makefiles or during a build.

Note The names and variables should be used as listed. Items written as $(NAME) are makefile variables and should also be used as-is.

`$(SUBDIR_CLEAN)`	This should be used as a dependency of the “clean” target in a component makefile. This will ensure that sub-libraries are also cleaned.
`$(SUBDIR_LIBS)`	A list of sub-libraries built by the make system. This is only set when `SUBDIRS` is assigned a value (see Section 6.9.2). This should be listed as a dependency of an object or library.
`<target>_DEBUG_RUN`	Same as <target>_RUN, except that it runs the target in the debugger. The naming convention is the same.
`<target>_RUN`	This can be used as a “make” target. The <exe target> should be replaced by the name of the target to run. For example, if myapp.lku is the executable target, myapp_RUN is the pseudo target to be used.
`clean_all`	This can be used as a “make” target to clean the component, all imported libraries and the `DVD_EXPORTS` directory (if defined).
`clean_imports`	This can be used as a “make” target to clean the imported libraries only.
`clean_libs`	This can be used as a “make” target to clean the `DVD_EXPORTS` directory (if defined).
`debug`	Same as the run target, except it runs the target in the debugger. This is not supported in version 1 makefiles.
`run`	This can be used as a “make” target to run the first executable target in the `ST20_TARGETS` or `ST40_TARGETS` or `ST200_TARGETS` list. This is not supported in version 1 makefiles.

10 Make System Macros

This section provides reference for the macros defined by the make system for use in component makefiles.

`BUILD_LIBRARY`	This macro defines the process used to create a library from list of object files and other libraries.
`COMPILE_C`	This macro defines the process used to create an object file from a source file (`*.c`). It should never be necessary to use this macro because the make system defines automatic rules for converting source files into object files.
`LINK_EXECUTABLE`	This macro defines the process used to create an executable from a set of object files and libraries. For ST20, this macro requires the definition of an executable entry point, of the form: *`executable`*`_INIT := `*`entry_point`* See Figure 5 for an example.
`ST20EXE_TO_ST40EXE`	A macro used to convert the ST20 executable naming convention to the ST40 equivalent. See notes on Section 12.2.4 for use of this macro.
`ST20LIB_TO_ST40LIB`	A macro used to convert the ST20 library naming convention to the ST40 equivalent.
`ST20OBJ_TO_ST40OBJ`	A macro used to convert the ST20 object naming convention to the ST40 equivalent.
`ST40EXE_TO_ST20EXE`	A macro used to convert the ST40 executable naming convention to the ST20 equivalent.
`ST40LIB_TO_ST20LIB`	A macro used to convert the ST40 library naming convention to the ST20 equivalent.
`ST40OBJ_TO_ST20OBJ`	A macro used to convert the ST40 object naming convention to the ST20 equivalent.
`ST20EXE_TO_ST200EXE`	A macro used to convert the ST20 executable naming convention to the ST200 equivalent.
`ST20LIB_TO_ST200LIB`	A macro used to convert the ST20 library naming convention to the ST200 equivalent.
`ST20OBJ_TO_ST200OBJ`	A macro used to convert the ST20 object naming convention to the ST200 equivalent.
`ST200EXE_TO_ST20EXE`	A macro used to convert the ST200 executable naming convention to the ST20 equivalent.
`ST200LIB_TO_ST20LIB`	A macro used to convert the ST200 library naming convention to the ST20 equivalent.
`ST200OBJ_TO_ST20OBJ`	A macro used to convert the ST200 object naming convention to the ST20 equivalent.

11 Common Makefile Errors

The following common errors should be avoided in new makefiles:

- Makefiles should be named "`makefile`" and not "`Makefile`".
- Makefiles should only build targets for files and directories that exist in that directory.

  This allows makefiles to avoid using relative paths which often cause failures. It also makes it easier to recognize which makefile applies to which set of files. This means that the following would not be allowed:

  ```
  OBJS := src/one.tco src/two.tco
  ```

  The side-effect of this is that a few more intermediary makefiles are required (especially in components that have a `src` directory), but in the long-run it makes for makefiles which are easier to understand.

- A makefile should not define "`-g`" flags to build debug versions of object files. This is achieved by issuing a "make" like this:

  ```
  make DEBUG=1
  ```

  The make system will issue the appropriate command to build debug object files necessary for running in the debugger.

  To limit the debug objects to components of interest, make a non-debug version of an executable; the components that should be debugged can be cleaned and the executable rebuilt with "`DEBUG=1`".

- Library makefiles should not set a `-DSTTBX_PRINT` as a `CFLAG`. Doing so would result in any toolbox print commands being included in the library, which is not desirable. When a component library is built, it should be non-debug (no "`-g`") and not have any toolbox print commands included.

- Within the development environment, symbolic file links should be avoided. This has traditionally been used to make a header file (like the exported component header file) appear in the component root directory.

  As an example, STSUBT (at the time of writing) defines `stsubt.h` in the `src/api` subdirectory of `stsubt`. This is linked to the `stsubt` directory by means of a symbolic link. The preferred way of doing this is to move the `stsubt.h` to the root of `stsubt`, and then each part of `stsubt` can define the following line in the makefile:

  ```
  HEADER_IMPORTS += stsubt
  ```

  This will add the `stsubt` component directory to the include path, so the header file can be located.

  Symbolic file links can be used in "personalized" include directories which are tailored for a particular release. The only reason that this is allowed is because the files that they refer to are removed from their original location during the release process. For example, in the `dbref` release, all STAPI header files (`staud.h`, `stvid.h`, etc) are used in the include directory and removed from the component directory. This rule exists to ensure that header files are not duplicated within the tree.

- Components should only import those STAPI components that they use. This is important because it can be used as a way to ascertain the knock-on effect of library changes.

- Components should always list **all** STAPI components that they use. Components commonly omit "HEADER_IMPORTS" like stsys or regular imports like stcount and sttbx.

  It is sometimes necessary to import components used by imported components. For example, sttbx needs stuart and stpio, because these are included in the sttbx header file. To test whether an import list is complete, the DVD_INCLUDE directory can be left unset within a development tree - the make system will use header files from their component directory.

12 Appendices

12.1 Glossary of Make System Terms

Term	Explanation
build	The process of invoking the “make” command to build the desired object, library or executable.
build directory	Targets are being built for this directory.
build host	The machine that the build process is run on. This is invariably a Solaris or Microsoft Windows machine with the appropriate build tools installed.
compile flags	Extra parameters passed to a compile. (e.g. `-DMY_DEFINE=1`)
component	A group of functions that form part of the STAPI. These functions are focussed on manipulating a particular bit of hardware or achieving a particular goal within the STAPI. (e.g. STPTI)
development tree or environment	The internal ST development tree, maintained within a clearcase VOB.
executable	A file which can be loaded on the target platform and run. (e.g. `ptsapp.lku`)
exports	Generally, header files and libraries produced by a component for other components or executables to use. (e.g. STPIO exports `stpio.h` and `stpio.lib`)
flags	See compile flags or link flags.
HAL	Hardware Abstraction Layer
imports	One or more components which are required by a particular component. (e.g. STUART imports STPIO and STCOMMON)
in-place include	Rather than using a central include directory, the STAPI header files are extracted from their location in the STAPI tree. This should only by used in a development environment (a VOB build). Failure to set `DVD_INCLUDE` will result in the make system using in-place includes.
in-place libraries	When the make system does not export the libraries (when the architecture-specific exports directory and the `DVD_EXPORTS` variables are not set), it will search for libraries within their component directories.
in-place link	See in-place libraries.
include path	A path searched for a header file during a compile.

Table 1 : Glossary Of Make System Terms

Term	Explanation
installed tree or installation environment	The structure and environment of an installed system - usually associated with installation at a customer site.
library	A single file which has been created by combining a one or more object files. (e.g. `stuart.lib`)
library path	A path searched for an object or library file during a link or while creating a library.
link flags	Extra parameters passed to a link (e.g. `-M xyz.map`)
link path	See library path.
multi-architecture	Applying to more than one hardware architecture. Within the context of this document it refers to support for ST20 and ST40 builds.
object	An object file produced by compiling a source file. (e.g. `demo.tco`)
object directory	A new feature in the version 2 make system. For all makefiles that support it, all targets built by the make system will be stored in an objs/architecture directory within the build directory. (e.g. for an ST20 platform, the object directory is `objs/ST20`)
path	A list of directories to be searched for a particular file.
platform	The target hardware for the build. (e.g. mb282b)
sub-library	A library built as part of a component and is generally not exported, but forms part of a higher level executable or library. Quite often this sort of library is built within a sub-directory of a component.
targets	An object, library, executable or operation that is required to be built by the make system. (e.g. the default target for STEVT is `stevt.lib`)

Table 1 : Glossary Of Make System Terms

12.2 Makefile Templates

Makefile templates can be found in the `templates` subdirectory of the `make` VOB.

12.2.1 ST20 Component Makefile

```
# Sample ST20 makefile

DVD_MAKE_VERSION := 2
ifdef IN_OBJECT_DIR

include $(DVD_MAKE)/generic.mak

HEADER_IMPORTS := <components with header files only>
IMPORTS := <imported components>
EXPORTS := <exported library and header>
TARGETS := <exported library>

OBJS := <objects in exported library>

include $(DVD_MAKE)/defrules.mak

<exported library>: $(OBJS)
    @$(ECHO) Building $@
    $(BUILD_LIBRARY)

clean:
    @$(ECHO) Cleaning $(TARGETS)
    -$(RM) $(OBJS)
    -$(RM) $(TARGETS)

# Local header dependencies

else

include $(DVD_MAKE)/builddir.mak

endif

# End of sample ST20 makefile
```

Template name: `ST20_component_makefile`

This is a makefile to build a simple library for an ST20. Use the ST20/ST40 makefile template when both architectures are to be supported. Replace all sections of the makefile within angle braces (“<“ and “>”). Insert any local header dependencies after the header dependency comment.

12.2.2 ST40 Component Makefile

```
# Sample ST40 makefile

DVD_MAKE_VERSION := 2
ifdef IN_OBJECT_DIR

include $(DVD_MAKE)/generic.mak

HEADER_IMPORTS := <components with header files only>
ST40_IMPORTS := <imported components>
ST40_EXPORTS := <exported library and header>
ST40_TARGETS := <exported library>

OBJS := <objects in exported library>

include $(DVD_MAKE)/defrules.mak

<exported library>: $(OBJS)
    @$(ECHO) Building $@
    $(BUILD_LIBRARY)

clean:
    @$(ECHO) Cleaning $(ST40_TARGETS)
    -$(RM) $(OBJS)
    -$(RM) $(ST40_TARGETS)

# Local header dependencies

else

include $(DVD_MAKE)/builddir.mak

endif


# End of sample ST40 makefile
```

Template name: `ST40_component_makefile`

This is a makefile to build a simple library for an ST40. Use the ST20/ST40 makefile template when both architectures are to be supported. Replace all sections of the makefile within angle braces (“<“ and “>”). Insert any local header dependencies after the header dependency comment.

12.2.3 ST200 Component Makefile

```
# Sample ST200 makefile

DVD_MAKE_VERSION := 2
ifdef IN_OBJECT_DIR

include $(DVD_MAKE)/generic.mak

HEADER_IMPORTS := <components with header files only>
ST200_IMPORTS := <imported components>
ST200_EXPORTS := <exported library and header>
ST200_TARGETS := <exported library>

OBJS := <objects in exported library>

include $(DVD_MAKE)/defrules.mak

<exported library>: $(OBJS)
    @$(ECHO) Building $@
    $(BUILD_LIBRARY)

clean:
    @$(ECHO) Cleaning $(ST200_TARGETS)
    -$(RM) $(OBJS)
    -$(RM) $(ST200_TARGETS)

# Local header dependencies

else

include $(DVD_MAKE)/builddir.mak

endif


# End of sample ST200 makefile
```

Template name: `ST200_component_makefile`

This is a makefile to build a simple library for an ST200. Use the ST20/ST40/ST200 makefile template when all the architectures are to be supported. Replace all sections of the makefile within angle braces (“<“ and “>”). Insert any local header dependencies after the header dependency comment.

12.2.4 ST20/ST40/ST200 Component Makefile

```
# Sample ST20/ST40/ST200 makefile

DVD_MAKE_VERSION := 2
ifdef IN_OBJECT_DIR

include $(DVD_MAKE)/generic.mak

HEADER_IMPORTS := <components with header files only>

ST20_IMPORTS := <imported components>
ST20_EXPORTS := <ST20 exported library and header>
ST20_TARGETS := <ST20 exported library>

ST20_OBJS := <ST20 objects in exported library>


ST40_IMPORTS := <imported components>|$(ST20_IMPORTS)
ST40_EXPORTS := <ST40 exported library and header>
ST40_TARGETS := <ST40 exported library>

ST40_OBJS := <ST40 objects>|$(call ST20OBJ_TO_ST40OBJ,$(ST20_OBJS))


ST200_IMPORTS := <imported components>|$(ST20_IMPORTS)
ST200_EXPORTS := <ST200 exported library and header>
ST200_TARGETS := <ST200 exported library>

ST200_OBJS := <ST200 objects>|$(call ST20OBJ_TO_ST200OBJ,$(ST20_OBJS))


include $(DVD_MAKE)/defrules.mak

$(LIB_PREFIX)<exported library>$(LIB_SUFFIX): $($(ARCHITECTURE)_OBJS)
    @$(ECHO) Building $@
    $(BUILD_LIBRARY)

clean:
    @$(ECHO) Cleaning $($(ARCHITECTURE)_TARGETS)
    -$(RM) $($(ARCHITECTURE)_OBJS)
    -$(RM) $($(ARCHITECTURE)_TARGETS)

# Local header dependencies

else

include $(DVD_MAKE)/builddir.mak

endif


# End of sample ST20/ST40/ST200 makefile
```

Template name: `combined_component_makefile`

This is a makefile to build a simple library for a ST20, ST40 and ST200. Replace all sections of the makefile within angle braces (“<“ and “>”). Insert any local header dependencies after the header dependency comment. The following additional points also apply:

- HEADER_IMPORTS are applied to ST20, ST40 and ST200 builds and are not specified separately.
- The ST40_IMPORTS and ST200_IMPORTS may list different imports or may just assign ST20_IMPORTS to this variable if the import lists are identical.
- ST40_EXPORTS and ST40_TARGETS or ST200_EXPORTS and ST200_TARGETS could be restated to appropriately convert ST20_EXPORTS or ST20_TARGETS, respectively. Each could be stated in the following way:

```
ST40_EXPORTS := $(call ST20LIB_TO_ST40LIB,$(ST20_EXPORTS))
ST40_TARGETS := $(call ST20LIB_TO_ST40LIB,$(ST20_TARGETS))

ST200_EXPORTS := $(call ST20LIB_TO_ST200LIB,$(ST20_EXPORTS))
ST200_TARGETS := $(call ST20LIB_TO_ST200LIB,$(ST20_TARGETS))
```

  These will convert library naming from ST20 to ST40/ST200, leaving the header files listed in the exports unchanged. If the targets include any object files too, the following line should be appended:

```
ST40_TARGETS := $(call ST20EXE_TO_ST40EXE,$(ST40_TARGETS))
ST200_TARGETS := $(call ST20EXE_TO_ST200EXE,$(ST200_TARGETS))
```

- The ST40_OBJS and ST200_OBJS can list the component object files (e.g. “one.o two.o three.o”). If the list of files is identical (except for the change in object extension), the provided macro call can be used.
- The “clean” target is complex because it needs to support ST20, ST40 and ST200. If the readability is deemed to be compromised, the target can be replaced with the following:

```
clean:
ifeq “$(ARCHITECTURE)” “ST20”
    @echo Cleaning $(ST20_TARGETS)
    -$(RM) $(ST20_OBJS)
    -$(RM) $(ST20_TARGETS)
else
    ifeq ““$(ARCHITECTURE)” “ST40”
        @echo Cleaning $(ST40_TARGETS)
        -$(RM) $(ST40_OBJS)
        -$(RM) $(ST40_TARGETS)
    else
        @echo Cleaning $(ST200_TARGETS)
        -$(RM) $(ST200_OBJS)
        -$(RM) $(ST200_TARGETS)
    endif

endif
```

  While only the ST20, ST40 and ST200 architectures are supported, this construct can be used. Once new architectures are supported, further “ifeq” constructs should be added.

12.2.5 ST20 Test Directory Makefile

```
# Sample ST20 makefile

DVD_MAKE_VERSION := 2
ifdef IN_OBJECT_DIR

include $(DVD_MAKE)/generic.mak

HEADER_IMPORTS := <components with header files only>
IMPORTS := <imported components>
EXPORTS := <likely to be blank - nothing exported>
TARGETS := <list of .lku files to build>

<target>_OBJS := <objects in particular .lku target>
...

include $(DVD_MAKE)/defrules.mak

<target basename>_INIT := <entry point>
<target .lku>: $(<target>_OBJS) $(IMPORT_LIBS)
    @$(ECHO) Linking $@
    $(LINK_EXECUTABLE)
...

clean:
    @$(ECHO) Cleaning $(TARGETS)
    -$(RM) $(<target>_OBJS)
    ...
    -$(RM) $(TARGETS)

# Local header dependencies

else

include $(DVD_MAKE)/builddir.mak

endif

# End of sample ST20 makefile
```

Template name: `ST20_tests_makefile`

This is a makefile to build one or more executables for an ST20. Use the ST20/ST40 makefile template when both architectures are to be supported. Replace all sections of the makefile within angle braces (“<“ and “>”). Insert any local header dependencies after the header dependency comment.

The ellipses associated with the target `OBJS` variable, target rules and “`clean`” rule indicate that the items are repeated for each of the `.lku` targets listed in the `TARGETS` line.

For each target rule, an associated `<target basename>_INIT` variable must be defined which provides the `LINK_EXECUTABLE` macro with the executable entry point. For example, a target of `myapp.lku` could have the following definition:

```
myapp_INIT := board_init
```

12.2.6 ST40 Test Directory Makefile

```
# Sample ST40 makefile

DVD_MAKE_VERSION := 2
ifdef IN_OBJECT_DIR

include $(DVD_MAKE)/generic.mak

HEADER_IMPORTS := <components with header files only>
ST40_IMPORTS := <imported components>
ST40_EXPORTS := <likely to be blank - nothing exported>
ST40_TARGETS := <list of .exe files to build>

<target>_OBJS := <objects in particular .exe target>
...

include $(DVD_MAKE)/defrules.mak

<target .exe>: $(<target>_OBJS) $(IMPORT_LIBS)
    @$(ECHO) Linking $@
    $(LINK_EXECUTABLE)
...

clean:
    @$(ECHO) Cleaning $(ST40_TARGETS)
    -$(RM) $(<target>_OBJS)
    ...
    -$(RM) $(ST40_TARGETS)

# Local header dependencies

else

include $(DVD_MAKE)/builddir.mak

endif

# End of sample ST40 makefile
```

Template name: `ST40_tests_makefile`

This is a makefile to build one or more executables for an ST40. Use the ST20/ST40 makefile template when both architectures are to be supported. Replace all sections of the makefile within angle braces (“<“ and “>”). Insert any local header dependencies after the header dependency comment.

The ellipses associated with the target `OBJS` variable, target rules and “`clean`” rule indicate that the items are repeated for each of the `.exe` targets listed in the `TARGETS` line.

For ST40, a `<target basename>_INIT` variable is not required (unlike for the ST20 shown in the previous template).

12.2.7 ST200 Test Directory Makefile

```
# Sample ST200 makefile

DVD_MAKE_VERSION := 2
ifdef IN_OBJECT_DIR

include $(DVD_MAKE)/generic.mak

HEADER_IMPORTS := <components with header files only>
ST200_IMPORTS := <imported components>
ST200_EXPORTS := <likely to be blank - nothing exported>
ST200_TARGETS := <list of .exe files to build>

<target>_OBJS := <objects in particular .exe target>
...

include $(DVD_MAKE)/defrules.mak

<target.exe>: $(<target>_OBJS) $(IMPORT_LIBS)
    @$(ECHO) Linking $@
    $(LINK_EXECUTABLE)
...

clean:
    @$(ECHO) Cleaning $(ST200_TARGETS)
    -$(RM) $(<target>_OBJS)
    ...
    -$(RM) $(ST200_TARGETS)

# Local header dependencies

else

include $(DVD_MAKE)/builddir.mak

endif

# End of sample ST200 makefile
```

Template name: `ST200_tests_makefile`

This is a makefile to build one or more executables for an ST200. Use the combined makefile template when both architectures are to be supported. Replace all sections of the makefile within angle braces (“<“ and “>”). Insert any local header dependencies after the header dependency comment.

The ellipses associated with the target `OBJS` variable, target rules and “`clean`” rule indicate that the items are repeated for each of the `.exe` targets listed in the `TARGETS` line.

For ST200, a `<target basename>_INIT` variable is not required (unlike for the ST20 shown in the earlier templates).

12.2.8 ST20/ST40/ST200 Test Directory Makefile

```
# Sample ST20/ST40/ST200 makefile

DVD_MAKE_VERSION := 2
ifdef IN_OBJECT_DIR

include $(DVD_MAKE)/generic.mak

HEADER_IMPORTS := <components with header files only>

ST20_IMPORTS := <ST20 imported components>
ST20_EXPORTS := <likely to be blank - nothing exported>
ST20_TARGETS := <list of .lku files to build>

ST40_IMPORTS := <ST40 imported components>|$(ST20_IMPORTS)
ST40_EXPORTS := <likely to be blank - nothing exported>
ST40_TARGETS := <list of .exe files to build>

ST200_IMPORTS := <ST200 imported components>|$(ST20_IMPORTS)
ST200_EXPORTS := <likely to be blank - nothing exported>
ST200_TARGETS := <list of .exe files to build>

<target>_ST20_OBJS := <objects in particular .lku target>
<target>_ST40_OBJS := <objects in particular .exe target>
<target>_ST200_OBJS := <objects in particular .exe target>

include $(DVD_MAKE)/defrules.mak

<.lku target basename>_INIT := <entry point>
<target>$(EXE_SUFFIX): $($(ARCHITECTURE)_OBJS) $(IMPORT_LIBS)
    @$(ECHO) Linking $@
    $(LINK_EXECUTABLE)

clean:
    @$(ECHO) Cleaning $($(ARCHITECTURE)_TARGETS)
    -$(RM) $(<target>_$(ARCHITECTURE)_OBJS)
    -$(RM) $($(ARCHITECTURE)_TARGETS)

# Local header dependencies

else

include $(DVD_MAKE)/builddir.mak

endif

# End of sample ST20/ST40/ST200 makefile
```

Template name: `combined_tests_makefile`

This is a makefile to build one or more executables for a ST20/ST40/ST200. Use this ST20/ST40/ST200 makefile template when both architectures are to be supported. Replace all sections of the makefile within angle braces (“<“ and “>”). Insert any local header dependencies after the header dependency comment. The following additional points also apply:

- HEADER_IMPORTS are applied to ST20, ST40 and ST200 builds and are not specified separately.
- The ST40_IMPORTS or ST40_IMPORTS may list different imports or may just assign ST20_IMPORTS to this variable if the import lists are identical.
- ST40_EXPORTS and ST40_TARGETS or ST200_EXPORTS and ST200_TARGETS could be restated to appropriately convert ST20_EXPORTS or ST20_TARGETS, respectively. Each could be stated in the following way:

```
ST40_EXPORTS := $(call ST20EXE_TO_ST40EXE,$(ST20_EXPORTS))
ST40_TARGETS := $(call ST20EXE_TO_ST40EXE,$(ST20_TARGETS))

ST200_EXPORTS := $(call ST20EXE_TO_ST200EXE,$(ST200_EXPORTS))
ST200_TARGETS := $(call ST20EXE_TO_ST200EXE,$(ST200_TARGETS))
```

  This will convert executable naming from ST20 to ST40/ST200, leaving the header files listed in the exports unchanged.

- An appropriately named _OBJS variable is created (depicted generically) for each of the target executables and for the ST20, ST40 and ST200. Only one generalised example is given for ST20, ST40 and ST200, but it is implied that this be repeated for all target executables.
- The <target>_ST40_OBJS or <target>_ST200_OBJS can list the component object files (e.g. “one.o two.o three.o”). If the list of files is identical (except for the change in object extension), the following macro call can be used:

```
$(call ST20OBJ_TO_ST40OBJ,$(<target>_ST20_OBJS))
$(call ST20OBJ_TO_ST200OBJ,$(<target>_ST20_OBJS))
```

- Quite often in a test directory, a single object file is linked with one or more libraries to produce an new test application. In this case it is possible to replace the definition of the _OBJS variables and executable targets, with the following lines:

```
%.lku: %.tco $(IMPORT_LIBS)
    $(LINK_EXECUTABLE)
%.exe: %.o $(IMPORT_LIBS)
    $(LINK_EXECUTABLE)
```

  If test1.lku and test1.exe were the targets for an ST20 and ST40/ST200, the make system will try and locate test1.tco and test1.o, respectively. These would then be created by compiling test1.c.

- Entry points must be defined for each of the .lku targets. See Section 12.2.5 for more information.
- The “clean” target is complex because it needs to support ST20, ST40 and ST200. See comment in Section 12.2.4 for more information.

12.2.9 Makefile for Component with Src Directory

This makefile is recommended for components with their own `src` directory. It is a basic requirement of a component that it export an appropriately named header file and library in the root of the component directory. The following recommendations are made (for direction purposes, an ST20 target is assumed):

- The header file be placed in the root of the component directory (for STAUD, this means `staud.h` is in `dvdbr-prj-staud`).
- The sub-directories use the library without duplication within the tree - this can be achieved by adding staud to the `HEADER_IMPORTS` line of the subdirectory makefiles.
- The library (e.g. `staud.lib`) be built in the src directory. This means that the `staud.lib` be listed in `TARGETS`, but not in `EXPORTS` in the `src` directory makefile. See Section 12.2.10 for the makefile template used in this case.

The following makefile will then be the template for the root of the component directory (above the `src` directory). The makefile will support ST20, ST40 and ST200 builds.

```
# Sample ST20/ST40/ST200 makefile

DVD_MAKE_VERSION := 2
ifdef IN_OBJECT_DIR

include $(DVD_MAKE)/generic.mak

COMPONENT_NAME := <component name>
ST20_TARGETS := $(COMPONENT_NAME).lib
ST20_EXPORTS := $(COMPONENT_NAME).h $(COMPONENT_NAME).lib

ST40_TARGETS := $(call ST20LIB_TO_ST40LIB,$(ST20_TARGETS))
ST40_EXPORTS := $(call ST20LIB_TO_ST40LIB,$(ST20_EXPORTS))

ST200_TARGETS := $(call ST20LIB_TO_ST200LIB,$(ST20_TARGETS))
ST200_EXPORTS := $(call ST20LIB_TO_ST200LIB,$(ST20_EXPORTS))

include $(DVD_MAKE)/defrules.mak

FULLDIR := $(DVD_BUILD_DIR)/src/objs/$(OBJECT_DIRECTORY)

$($(ARCHITECTURE)_TARGETS): $(FULLDIR)/$($(ARCHITECTURE)_TARGETS)
    $(CP) $(subst $(BAD_SLASH),$(GOOD_SLASH),$<) $@ > $(NULL)

$(FULLDIR)/$($(ARCHITECTURE)_TARGETS): FORCE
    @$(ECHO) Entering SRC directory
    $(MAKE) -C $(DVD_BUILD_DIR)/src

clean: subdir_clean
    @$(ECHO) Cleaning $($(ARCHITECTURE)_TARGETS)
    $(RM) $($(ARCHITECTURE)_TARGETS)

subdir_clean:
    $(MAKE) -C $(DVD_BUILD_DIR)/src clean
```

```
FORCE:

else

include $(DVD_MAKE)/builddir.mak

endif

# End of sample ST20/ST40/ST200 makefile
```

Template name: `combined_toplevel_makefile`

Extra notes about this template:

- Only the value of `COMPONENT_NAME` needs to be set. So for STAUD, the line becomes:

  ```
  COMPONENT_NAME := staud
  ```

- No other changes should be required.
- The library will be copied from the src directory to the root of the component directory. The definition of `CP` in the toolkit usually preserves date and time stamps in the copy.

12.2.10 Component Src Directory Makefile

This template is related to the previous template. Whereas that makefile is suggested for the directory above the `src` directory, this template is used in the `src` directory when the directory contains a number of sub-libraries to be built in subdirectories.

```
# Sample ST20/ST40/ST200 makefile

DVD_MAKE_VERSION := 2
ifdef IN_OBJECT_DIR

include $(DVD_MAKE)/generic.mak

ST20_TARGETS := <ST20 target library>
ST40_TARGETS := <ST40 target library>
ST200_TARGETS := <ST200 target library>

SUBDIRS := <list of subdirs to build>

include $(DVD_MAKE)/defrules.mak

$(LIB_PREFIX)<target library>$(LIB_SUFFIX): $($(ARCHITECTURE)_OBJS)
    @$(ECHO) Building $@
    $(BUILD_LIBRARY)

clean: $(SUBDIR_CLEAN)
    @$(ECHO) Cleaning $($(ARCHITECTURE)_TARGETS)
    -$(RM) $($(ARCHITECTURE)_TARGETS)

# Local header dependencies

else

include $(DVD_MAKE)/builddir.mak

endif

# End of sample ST20/ST40/ST200 makefile
```

Template name: `combined_srcdir_makefile`

The following additional points apply:

- Replace all sections of the makefile within angle braces (“<“ and “>”).
- If there are a number of object files to be built in the `src` directory, they can be listed in variables named `ST20_OBJS, ST40_OBJS` and `ST40_OBJS`. Those variables are then added as dependencies of the ST20, ST40 and ST200 libraries respectively. These must also be added to the clean target.
- A common feature in this type of makefile is to add to `SUBDIRS` based on configuration options. See Section 6.12.4.
- The SUBDIRS can also vary based on the `ARCHITECTURE`. The following lines show an example of that:

  ```
  ifeq “$(ARCHITECTURE)” “ST20”
  ```

```
    SUBDIRS += one two
endif
```

12.2.11 Component level LINUX Makefile (linux.mak)

This makefile is recommended for components with linux directory. It is a basic requirement of a component that it is to be compiled on DVD_OS = LINUX.

The following makefile will then be the template for the root of the component directory

Sample linux.mak makefile present in top directory of component

```
DVD_MAKE_VERSION := 2

include $(DVD_MAKE)/generic.mak

.PHONY: build_all_linux

LINUX_TARGETS := build_all_linux

LINUX_EXPORTS := <exported library, .ko and header>

include $(DVD_MAKE)/defrules.mak


LINUX_EXPORTS_WITHOUT_HEADERS := $(filter-out %.h,$(LINUX_EXPORTS))


build_all_linux:
    @$(ECHO) Building all LINUX targets
    @$(MAKE) -C $(DVD_BUILD_DIR)/linux
    @$(CP) $(addprefix $(DVD_BUILD_DIR)/linux/objs/$(OBJECT_DIRECTORY)/
,$(LINUX_EXPORTS_WITHOUT_HEADERS)) .

clean:
    @$(ECHO) Cleaning $(DVD_BUILD_DIR)
    @$(RM) $(LINUX_EXPORTS_WITHOUT_HEADERS)
    @$(MAKE) -C $(DVD_BUILD_DIR)/linux clean
```

Template name: `STLINUX_component_makefile`

12.2.12 Makefile present in linux folder

This makefile is recommended for components with linux directory. It is a basic requirement of a component that it is to be compiled on DVD_OS = LINUX.

The following makefile will then be the template for the linux folder of the component directory

Sample makefile

```
DVD_MAKE_VERSION := 2
ifdef IN_OBJECT_DIR

include $(DVD_MAKE)/generic.mak

LINUX_TARGETS := <exported library and .ko >

include $(DVD_MAKE)/defrules.mak

# For building kernel objects
$(filter %.ko,$(LINUX_TARGETS)): FORCE
    @$(ECHO) Building $@
    -$(CP) $(DVD_MAKE)/Modules.symvers $(DVD_BUILD_DIR)/$(basename $@)/.
    @$(MAKE) -C $(DVD_BUILD_DIR)/$(basename $@)
    @$(CP) $(DVD_BUILD_DIR)/$(basename $@)/$@ .
    -$(RENAME) $(DVD_BUILD_DIR)/$(basename $@)/Modules.symvers $(DVD_MAKE)/
.


# For building libraries
$(filter lib%$(LIB_SUFFIX),$(LINUX_TARGETS)): FORCE
    @$(ECHO) Building $@
    @$(MAKE) -C $(DVD_BUILD_DIR)/$(patsubst lib%$(LIB_SUFFIX),%_ioctl,$@)
$@
    @$(CP) $(DVD_BUILD_DIR)/$(patsubst lib%$(LIB_SUFFIX),%_ioctl,$@)/$@ .

FORCE:

clean:
    @$(ECHO) Cleaning $(DVD_BUILD_DIR)
    @$(MAKE) -C $(DVD_BUILD_DIR)/<component>_core clean
    @$(MAKE) -C $(DVD_BUILD_DIR)/<component>_ioctl clean
```

Template name: `STLINUX_linux_folder_makefile`

12.2.13 Makefile present in linux/<component>_ioctl folder

This makefile is recommended for components with linux directory.

The following makefile is the template for the linux/<component>_ioctl folder of the component directory

Sample makefile

```
HEADER_IMPORTS += <components with header files only>

include $(DVD_MAKE)/kbuild.mak

<component>_ioctl-objs := <objects in exported library>

EXTRA_CFLAGS += $(DVD_INCLUDE_PATH)
EXTRA_CFLAGS += $(KBUILD_CFLAGS)
EXTRA_CFLAGS += $(DVD_LINUX_CFLAGS)


ifneq ($(KERNELRELEASE),)
# Kernel makefile
else
ifeq "$(KDIR)" ""
$(error The enviroment variable KDIR must be set)
endif

# External makefile
PWD     := $(shell pwd)

all: default <exported library>


default:
    $(MAKE) -C $(KDIR) M=$(PWD)  modules

<exported library>: <component>_ioctl_lib.o
    $(BUILD_LIBRARY)
```

Template name: STLINUX_ioctl_makefile

12.2.14 Makefile present in linux/<component>_core folder

This makefile is recommended for components with linux directory.

The following makefile is the template for the linux/<component>_core folder of the component directory

Sample makefile

```
HEADER_IMPORTS += <components with header files only>

include $(DVD_MAKE)/kbuild.mak

<COMPONENT>_OBJS := <objects in src folder>

obj-m           := <component>_core.o
<component>_core-objs := <objects in exported library>\
          $(<COMPONENT>_OBJS)

EXTRA_CFLAGS += $(DVD_INCLUDE_PATH)
EXTRA_CFLAGS += $(KBUILD_CFLAGS)

# The following checks to see if we have been invoked in the kbuild
# (KERNELRELEASE will be defined). If not we have the means of launching
# the KBUILD (all and default targets).

ifneq ($(KERNELRELEASE),)

# Kernel makefile

else

ifeq "$(KDIR)" ""
$(error The enviroment variable KDIR must be set)
endif

PWD := $(shell pwd)

all: default

default:
    $(MAKE) -C $(KDIR) M=$(PWD)  modules

# Remove the object files, the .<object>.cmd file and use KBUILD to remove
the rest
clean:
    $(RM) $$(<COMPONENT>_OBJS)
    $(RM) $(foreach FILE,$(<COMPONENT>_OBJS),$(dir $(FILE)).$(notdir
$(FILE)).cmd)
    $(MAKE) -C $(KDIR) M=$(PWD) clean

endif
```

Template name: `STLINUX_core_makefile`