MoReUse / SDE2 2.3
User Manual Version 3.8

User Manual Version 3.8

Sep 29, 2006

STA/SDM/SDE2_2.3/0007

User Manual Version 3.8
Publication Date: Sep 29, 2006

Abstract

This document is the User's Manual of SDE2 Version 2.3.

Keywords

MoReUse, SDE2, Users' Manual

References

[MoReUse]MoReUse 3.1 Standards Book

LIPP/2005/055 - December 2005

[RELNOT]DVP SDE2 2.3 Release Notes,

STA/SDM/SDE2_2.3/0012

[RULES]MoReUse Rules Document

RTG/CMD/2001/0252, version 2.10 – October 2005

[QMORE]Qmore 03.01.00 User Manual, version 1.0,

SDM/ReDT/Software/Qmore_03.01.00/018

[BCaM]http://pww.cto.sc.philips.com/bcam-processes/

Revision History

2003-04-16	2.2	SDE2 1.7 Alpha	Proposed	Shivaraj P, Bhaskar G
2003-04-22	2.3	SDE2 1.7 Beta	Accepted	Shivaraj P
2003-06-16	2.4	SDE2 1.7	Approved	Bhaskar G
2004-05-19	2.5	SDE2 2.0 Alpha	Proposed	Shivaraj P
2004-06-18	2.6	SDE2 2.0 Beta	Accepted	Shivaraj P
2004-07-12	2.7	SDE2 2.0	Approved	Shivaraj P
2004-03-18	2.8	SDE2 2.1 Alpha	Proposed	Shivaraj P,
2004-04-08	2.9	SDE2 2.1 Beta	Accepted	Shivaraj P
2004-05-20	3.0	SDE2 2.1	Approved	Bhaskar G
2005-09-02	3.1	SDE2 2.1_SP1	Approved	Bhaskar G
2005-09-23	3.2	SDE2 2.2 Alpha	Proposed	Roopa M
2005-09-30	3.3	SDE2 2.2 Beta	Accepted	Bhaskar G
2005-10-28	3.4	SDE2 2.2	Approved	Bhaskar G
2006-06-02	3.5	SDE2 2.2_SP1	Approved	Bhaskar G
2006-07-28	3.6	SDE2 2.3 Alpha	Proposed	Bhaskar G
2006-08-11	3.7	SDE2 2.3 Beta	Accepted	Bhaskar G
2006-09-29	3.8	SDE2 2.3	Approved	Bhaskar G

Introduction 1

Tutorial 5

Reference Manual 10

Chapter 1
Introduction

User Manual Version 3.8 **Sep 29, 2006**

What is covered in this chapter?

This chapter provides you with basic background information about SDE2, including:

- Description of the SDE2 directory structure
- Explanation of the basic principles of SDE2

1.1 Prerequisites

This manual assumes you are familiar with:

- C programming language
- Perl programming language
- Makefiles
- gmake

  For more information about **gmake** see:

  http://www.gnu.org/manual/make/html_mono/make.html

Please familiarize yourself with the terms and abbreviations in the *Glossary* before reading this manual.

1.1.1 Tools and Softwares Required

SDE2 requires the following tools and softwares installed on the system, other than the compiler toolsets

- Perl - The latest version can be downloaded from:

  http://www.perl.com/download.csp

- Cygwin (for windows hosts only)- SDE2 delivers cygwin for windows users along with the release. However, the latest version of cygwin can be downloaded from:

  http://www.cygwin.com/

- GNU make (gmake) - SDE2 delivers gmake for windows users along with the release. Other users can download the latest version of gmake from:

  http://directory.fsf.org/GNU/make.html

- Doxygen and Graphviz - SDE2 users who need to generate Auto Documentation can download Doxygen and Graphviz from:

  Doxygen - http://www.doxygen.org/ or http://www.stack.nl/~dimitri/doxygen/

  Graphviz- http://www.graphviz.org/

1.2 Purpose and Scope

SDE2 is a MoReUse-compliant software development environment created for use by both NXP Semiconductors developers and external customers. SDE2 allows you to:

- Build for multiple configurations
- Integrate third-party tools at build time
- Build binary releases
- Create reusable software

SDE2 is a generic, integrated and MoReUse compliant software build environment for the production of reusable software IPs and systems.

The term generic in the context of build implies the support for multiple configurations.
The term integrated in the context of build implies the support for integrating 3rd party tools.
Ther term production means the support for producing binary releases.
The term reusable means the support for reusable software development.

The above common approach enables better reuse of each other's software components. In this context, SDE2 is a MoReUse compliant directory structure with supporting makefiles and build scripts. These supporting makefiles execute dependency checking, compilation and linking for a variety of platforms and a variety of compilation hosts. The makefiles are executed using GNU Make. SDE2 supports four host platforms: PC/Windows and Linux.

SDE2 supports component-based working for C/C++, System C and Java environments.

SDE2 is compliant with the directory structure, file names and rules of MoReUse[1], see *[MoReUse] MoReUse 3.1 Standards Book* in the bibliography.

1.3 Directory Structure

SDE2 and MoReUse implement a flat directory structure. This decision was made because:

- Directory structure is independent of owner, project, and architecture.

  This makes components more re-usable (outside the scope of their development environment). An architectural split could be along several axes: functional grouping (infra, gfx, video, audio, mux-demux, storage) or along layering (hwapi, devlib, tssa, sub-systems).

- All visible components are separately released.

  There are many choices, and it is not always clear where a component belongs in architecture (tuner, psi-database, html-renderer). Components are released as entity.

1. MoReUse standards have been defined for C/C++ software. They have not yet been defined for Java. In SDE2, the same approach is used for Java sources as for C/C++ sources with minimal differences.

- The SDE2 can easily find the interface of components, and can thus enforce certain rules.

  The rules the SDE enforces are of strict interfacing. A component is required to explicitly state its dependencies on other components, and the SDE can enforce this.

Drawbacks of a flat directory structure can be:

- Too many components in one directory.

  It is unlikely that within a project there are, for example,100 components. This would be a lot to have in one directory, but by using different prefixes (**tm2d**, **tmdl**) there is some layering or subsystem grouping.

- Lost clarity due to hierarchy (sub-systems would like to have encompassed components below them in one component).

  This argues that the components are not real components in the sense that they have their own life cycle (independent of the subsystem life cycle). When this happens, it is argued that the encompassed components are not really components in terms of SDE2. The subsystems are then the component and beneath this component the developer is free to choose the directory structure.

  However SDE2 supports two extensions of the flat directory structure – multiproject development (you have multiple **`comps`** directories) and SDE_in_SDE (you have subcomponents in your **`src`** directory). These extensions do not contradict to the flat directory structure, you can always put all your components in one **`comps`** directory.

1.4 SDE2 basic principles

SDE2 is an associated product of SoCDT/LIPP's MoReUse Program. It implements the recommended software development approach for the software development community of NXP Semiconductors. It offers an off-the-shelf, MoReUse-compliant directory structure, with supporting makefiles and scripts.

When a user installs SDE2, it automatically creates the required directory structure and hence, any component developed and built with SDE2 environment will have the MoReUse directory structure.

The basic aim of SDE2 is to enable the creation and delivery of reusable software components, which can be used across different target platforms without any changes. A source component can be built for different platforms or configurations, by setting a corresponding set of platform-specific environment variables in the host environment.

SDE2 defines a configuration as a specific combination of CPU class, OS class and build toolchain. This is accomplished in the following ways:

- A user can develop reusable components for a particular configuration. This component can be reused across different development sites with out making any changes to the component itself. The user needs to release only the component.
- A user can develop reusable components for multiple configurations, by properly organizing the configuration-specific parts of the source code separately. These components are also reused across different development sites for different configurations.
- SDE2 will generate libraries and DLLs, whose names are MoReUse-compliant.

SDE2 also facilitates compilation and linking of component source code and provides the following facilities to the user:

- A user can define and use his/her own component or interface.
- A user can develop a component in a modular and independent way. A component is open to the external world only by its name and interfaces (public header files). Hence, these components are easily reusable in other applications (which require this component) just by addressing their interfaces.
- A user can build a complete application (executable), without knowing about the components required to build that application. SDE2 automatically builds all required components (in correct order) and then builds the application.
- A user can completely build a component for all specified configurations of the project.

The procedures and methodologies for the above features are explained in the subsequent sections of the SDE2 User Manual.

Chapter 2
Tutorial

User Manual version 3.8 **Sep 29, 2006**

What is covered in this chapter?

This chapter describes how to complete basic tasks such as compiling libraries and images, using an enclosed example, including:

- Creating and naming components
- Creating the directory structure
- Creating source and header files, and a makefile
- Building libraries and executables

2.1 Overview

This chapter is a tutorial for SDE2. We provide a short example to demonstrate SDE2. From a developer's point of view, the SDE2 is a directory structure you can put your software in. Using the SDE2 you are able to build your software for a variety of platforms.

In SDE2, 18 example components and two interfaces are provided. These components demonstrate different approaches to build libraries, DLLs, JARs and executables.

For the tutorial, the following is assumed:

- The SDE2 is installed on your PC or workstation
- At least one compiler is available (our example uses the TriMedia compiler)
- At least one software component is available in SDE2 (our example uses the component **Comp1**).

2.2 Creating a component

Creating a component is a standard procedure. The following section explain the necessary steps.

2.2.1 Obtaining a unique component name

Register your component on the MoReUse web site; for more details read *Section 3.1.8, Component names* on page 13.

2.2.2 Creating the directory structure

After creating the **tm<your component name>** directory in the main **comps** directory, you have to create its subdirectories and files. This includes the **inc**, **src**, **docs**, **tst** subdirectories, **makefile** and **configurations.txt** files. More information about the directory

structure can be found in *Section 3.1, Common directory structure* on page 10, and in *[MoReUse] MoReUse 3.1 Standards Book* in the bibliography. Put public header files in the `inc` directory and source and private header files in the `src` directory. Generally, your directory structure should look like the structure below

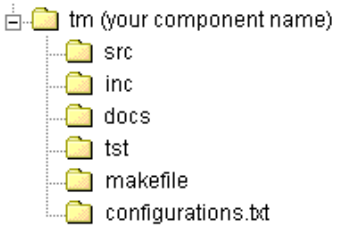


Figure 2-1: Structure of tm<your component name> directory

2.2.3 Creating the source and header files

This section contains simple example files for `src` and `inc` directories, that together form the C-part of a component.

The source file **src/tmComp1.c** containing the implementation of the interface is as follows:

```
/* This is tmComp1.c */
#include "tmComp1.h"
#include <stdio.h>

void tmComp1_Print(int i)
{
        printf("This is component 1, printing value %d\n", i);
}
```

Directory `src` contains source files and non public header files.

The file **inc/tmComp1.h** containing the public interface of the component **Comp1** is as follows:

```
/* This is tmComp1.h */
#ifndef TMCOMP1_H
#define TMCOMP1_H
extern void tmComp1_Print(int i);
#endif
```

The directory **inc** contains the public interface.

2.3 Creating a makefile

All component makefiles have a similar structure but there are some differences. They all are using some environment and makefile variables.

Every tutorial book starts with a simple **Hello world** example. Our component **Comp1** is such an example. This is its makefile:

```
DIR_LOCA L= comps/tmComp1
include $(_TMROOT)/sde/environment.mk
CXX_SOURCES =
C_SOURCES = src/tmComp1.c
all: configuration lib
ifneq ($(DIR_CONFIG),_)
include $(DIR_SDE)/$(DIR_CONFIG)/makelib.mk
endif
```

This is a simple component, it contains only the **src/tmComp1.c** source code and no diversities (see *Section 3.5, Diversities* on page 36 for more information). It has two targets. The **configuration** target is mandatory; it sets the configuration. The **lib** target builds the library.

The first line must contain the location of the component:

```
DIR_LOCAL= comps/tmComp1
```

The second line includes a file **environment.mk** that contains a number of settings that are general for the installation (i.e., independent of the component).

The following lines list the source files of the component; **CXX** stands for C++.

```
CXX_SOURCES =
C_SOURCES = src/tmComp1.c
```

If we build a library, we have to add a **lib** target in **all** targets.

After all the component settings are completed, we call the platform-specific **makelib.mk** file. This file takes care of making the libraries.

The makefile contains other relevant lines (These lines are typically modified when executing a makefile for a new component). If a component requires other component interfaces, you can use the line below to indicate that. If **Comp1** required the **tmosal** and **tmml** interfaces you would see the following line:

```
REQUIRES = tmCom tmCommMgr
```

If your component calls functions from another component, you must have line like this:

```
LIBS = tmCopyIO tmAencAc3
```

More information about makefiles can be found in *Section 3.15, Component makefile manual* on page 85 and elsewhere in *Chapter 3*.

2.4 Building a component (library)

When we make our component **tmComp1**, the build environment can be initialized. We open a shell (DOS or UNIX/Linux) to execute an initialization script. A default script named **tm_psos_debug_static_el_tm32_winnt_default.bat** is included in **`project/sites/<site>`**. The directory **`<site>`** contains site-specific information about SDE2. With SDE2 delivery,

there is a part with exemplary files in the **<site>** directory called **blrsdm**. See *Section 5.2.3, Tuning the SDE initialization script* on page 110 for information about making your site-specific initialization script.

The script sets some environment variables indicating the location of the compilers and so on. Some variables also indicate the build flavor[2].

The **tmComp1** directory contains a makefile that specifies how the component is built. The makefile lines are like the lines in the **Hello world**–type example in *Section 2.3, Creating a makefile* on page 6 with additional comment lines.

When the makefile is executed by typing **make** (or **gmake** for UNIX), the result is the creation of the following directory structure. The root location of the directory structure is configurable and depends on an environment variable **_TMTGTBUILDROOT**, (set in one of the batch files from **sde_template/project/sites/<your site>** directory, for example, the **tm_psos_debug_static_el_tm32_winnt_default.bat** script).

When **_TMTGTBUILDROOT** contains for example, **Build_Directory**, the directory structure will be as follows:

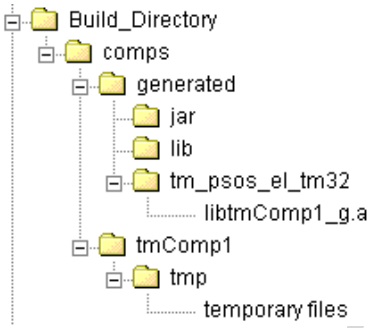


Figure 2-2: Structure of Build_directory

More information about the library directory structure can be found in *Section 3.1.11, Libraries location* on page 15.

2.5 Building an image (executable)

Building an image, for example a test application, is not much different from building a component. The difference is in the makefile. The image makefile always contains all first-level used libraries, the libraries with functions are directly called from the executable:

LIBS = tmComp1

The order of libraries is significant for UNIX-like linkers such as **mips_psos**, but SDE2 accounts for this. In the image-makefile there are also the following specific lines (in italics):

TARGET = test
all: configuration *target*

2. 'Flavor' is a property of a buildable object that can be separately defined. More information regarding flavors can be found in sections Section 3.5.1 and Section 3.13.

```
ifneq ($(DIR_CONFIG),_)
include $(DIR_SDE)/$(DIR_CONFIG)/maketarget$(_TMBSL).mk
endif
```

An example makefile of a simple executable that tests the component **Comp1** can be found in **`comps/tmComp1/tst/Tst1`** of the SDE2 distribution. After executing the makefile, the following structure is created:

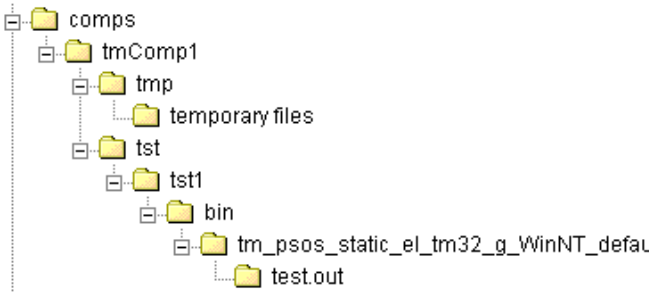


Figure 2-3: Structure after executing the makefile

The **test.out** file is a debug executable that can be downloaded on your TriMedia board.

Chapter 3
Reference Manual

User Manual Version 3.8 **Sep 29, 2006**

What is covered in this chapter?

This chapter contains comprehensive reference material about SDE2, including:

- Description of the common directory structure
- Selecting and identifying configurations
- Precompile flags and compile options
- Diversities
- Dynamic Link Libraries (DLLs)
- Transitive closure of libraries
- SDE-in-SDE
- Multiproject SDE2
- Generating binary releases
- Build scripts and flavors
- Component makefile manual

3.1 Common directory structure

SDE2 is component-based. A software component contains source files (including private header files), public header files (the API of the component), documentation, libraries, and test applications or executables. A software component that binds together other components (also called subsystems) is also a component and is treated in the same way as other components.

The SDE2 root directory contains the following directories:

- `inc`
- `sde`
- `project`
- `install`

The following directories are typically placed in the SDE2 root directory, however they may be placed in another location in case SDE2 is used in multiproject mode (see *Section 3.9.1, Multiproject implementation in SDE2* on page 57).

- `comps`
- `intfs`

- **apps**

The directories are explained in more detail below.

3.1.1 inc directory

The **inc** directory contains information that is applicable to all components. Types of information are:

- Global include files, such as **tmNxTypes.h**, **tmSystemFormats.h**, **tmAudioFormats.h**, **tmVideoFormats.h**, **tmCompId.h**, and **tmAvFormats.h**. The old **tmtypes.h** is no longer supported.
- Platform-specific directories such as **mips_psos**, **tm_psos**, **x86_nt**, **cfg**.

Do not modify the **inc** directory, see *[RULES] MoReUse Rules Document*. You can modify the configuration files in its subdirectories.

3.1.2 sde directory

The **sde** directory contains generic makefiles. Usually, it is not necessary for developers to know anything about the structure of this directory. Do not modify this directory. If you find something that needs to be changed, submit a change request to the SDE2 CCB, see *[MoReUse] MoReUse 3.1 Standards Book*. External SDE2 users should raise change requests through their contacts within NXP Semiconductors

3.1.3 project directory

The **project** directory contains the following files:

- **configurations.txt** – Contains all buildable configurations; see *Section 3.13.3.3, The overall configurations.txt file* on page 70 for more details.
- **buildlist.txt** – Contains the list of all components that have to be built.
- **prjlist.txt** – Contains all required project directories, separated by spaces. For details see *Section 3.9.1, Multiproject implementation in SDE2* on page 57.

The files **loc_list.txt** and **loc_list.mk** are derived from **prjlist.txt**. These files have been placed in the release directory since SDE2 version 1.2. For more information, see *Section 3.9, Multiproject SDE2* on page 56.

3.1.4 install directory

The **install** directory contains the files required for installting SDE2 in cadenv environment. Following are contents of this directory:

- **sde2_cadenv.hlp-** cadenv helpfile
- **sde2_cadenv.rel-** cadenv release file
- **sde2_cadenv.installnotes.txt-** cadenv SDE2 installation notes
- **sde2_cadenv.releasenotes.txt-** cadenv SDE2 release notes
- **sdebuild-** wrapper script for build.pl
- **sdedoc-** wrapper script for reading SDE2 documents

- **sdebuild_exe-** wrapper script for build_exe.pl
- **sdemake-** wrapper script for gmake

Please refer to *Section 5.3, SDE2 on cadenv* on page 112 for more information on cadenv support in SDE2

3.1.5 comps directory

All components are stored in the **comps** directory. This is the place to add new components. Restrictions and procedures regarding component names are described in *Section 3.1.8, Component names* on page 13.

Figure 3-1 illustrates the internal structure of the **comps** directory for the component **Comp1**.

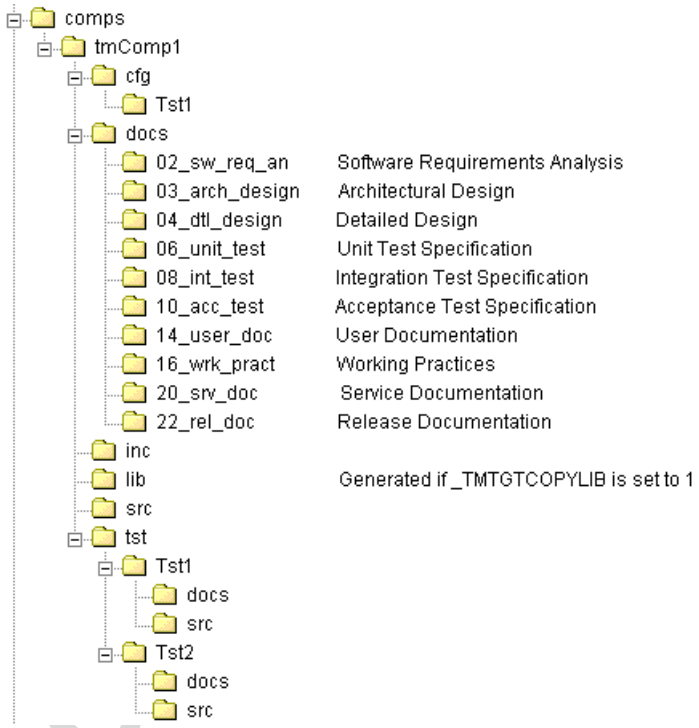


Figure 3-1: Internal structure of the comps directory

The **src** directory contains the source files and private header files. You can structure the **src** directory in any way. A specific structure in the case of a component with implementations on more than one platform, is suggested in *Section 3.1.10, Platform-specific source files* on page 14.

The **cfg** directory is optional. It contains configurational source and header files. These files are delivered with the binary release and the user can reconfigure and recompile his library files.

The **inc** directory contains the public header files of the component. The contents of the public header files form the public API of the component. The public header files should not have includes of other header files except for **tmNxTypes.h**, **tmSystemFormats.h**,

tmAudioFormats.h, tmVideoFormats.h, **tmCompId.h** and/or **tmAvFormats.h**. In this way large include trees are avoided, which shortens compilation time, and improves independent deployment of the components.

The **docs** directory contains the documents about the component. The documents are organized in the categories according to the development phases as specified in the Business Creation and Management Process document *[BCaM] http://pww.cto.sc.philips.com/bcam-processes/*

The **TstX** (in the example X = 1 or X = 2) directories contain the test application sources for the component. When a component contains more than one test application, separate directories beneath the **tst** directory need to be made for each test application.

During the build process, we build all libraries in the release **lib** directory. The optional component **lib** directory is a copy of the originally generated **lib** directory.

3.1.6 intfs directory

See *Section 3.1.9, Interfaces* on page 14.

3.1.7 apps directory

The **apps** directory contains applications (executables) that use more than one component. Applications are like test programs. However they are beyond the scope of one particular component. The **apps** directory typically contains the nonreusable part of a software system.

Summarizing, SDE2 describes a directory structure for a component-based way of working. The structure of the non project-specific directories **comps**, **inc**, and **intfs** are important from a reuse point of view and should be the same for all projects that generate MoReUse-compliant software components.

3.1.8 Component names

Component names must consist of letters (a..z, A..Z) and digits (0..9) only. Other characters (underscores, hyphens etc.) are not allowed. Component names must begin with a capital (A..Z). A main directory of a component is named after the component with the prefix **tm** and after the layer's name, if present. You can find registered layer names in *[MoReUse] MoReUse 3.1 Standards Book* . Sometimes the component name (e.g., **tmComp4**, **tmdlComp5**) is used in this document instead of the real component name (e.g., **Comp4**, **dlComp5**), to provide a generic example of the name.

To avoid name clashes, each component must have a unique component name and a unique component ID (used in its return status, see *[MoReUse] MoReUse 3.1 Standards Book*). Read the Rules document *[RULES] MoReUse Rules Document* and request a unique component name on the MoReUse site at:
http://pww.cto.sc.philips.com/products/reuse_standards/html/component_names.html.

All components should be registered using the MoReUse web site.

3.1.9 Interfaces

The **intfs** directory contains *interfaces* (public header files) that are implemented by multiple components. This is useful when your project uses an interface-based way of working. In that case you can specify an interface and have, for example, two components that implement this interface. To avoid one interface definition being located at two places (the **inc** directories of the two components implementing the interface), the header files are located at a central place outside the scope of a particular component, the **intfs** directory. Defining interfaces and using them across various implementations implies that interfaces must be stable as soon as a first implementation exists. Do not modify this directory, see *[RULES] MoReUse Rules Document*. Figure 3-2 shows the structure of the **intfs** directory.

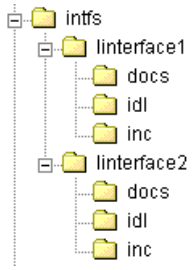


Figure 3-2: Structure of the intfs directory

Each interface name starts with **I**. The **intfs** directory possesses its own directory with the subdirectories **idl**, **inc**, **src** and **docs**.

The **src directory contains proxy and stub modules that are used when the component is present on another computer.**

The **idl** directory optionally contains the interface definition language (idl) description file(s) of the interface.

The **inc** directory contains the header file(s) of the interface.

The **docs** directory contains documentation describing the interface.

To avoid name clashes, each interface must have a unique interface name and a unique interface ID (used in its return status, see *[MoReUse] MoReUse 3.1 Standards Book*). Read the Rules document *[RULES] MoReUse Rules Document* and request a unique interface name on the MoReUse site at:
http://pww.cto.sc.philips.com/products/reuse_standards/html/interface_names.html.

For more about the promotion of the interfaces, see *Section 3.7.5, Promotion of the interface* on page 49.

3.1.10 Platform-specific source files

Sometimes, a component contains different source files for different configuration classes. This chapter describes the recommended way of working in this situation.

The platform-independent software is usually put in the main source directory (**src**). By using the environment variables _TMTGTCPUCLASS and _TMTGTOSCLASS in your makefile, you can select the source files for platform-specific software, for example,

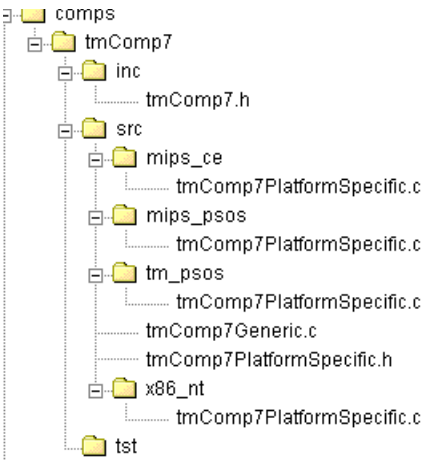


Figure 3-3: Example using platform-specific source files

The file **tmComp7PlatformSpecific.h** contains the (generic) prototypes of the functions that have platform-specific implementations.

In the makefile, the sources are defined as follows:

```
C_SOURCES=\
        src/ tmComp7Generic.c \
        src/$(_TMTGTCPUCLASS)_$(_TMTGTOSCLASS)/tmComp7PlatformSpecific.c
```

This approach works only if the source files have the same names in all directories. Otherwise you should use the defines from **tmFlags.h**. (See *Section 3.3, Standard precompile flags and tmFlags.h file* on page 29.)

3.1.11 Libraries location

The library structure is based on *[MoReUse] MoReUse 3.1 Standards Book*. In this section the following aspects are described:

- Library directory location
- Location of DLL files
- Location of JAR files

If _TMTGTBUILDROOT is not empty, the library directory structure is a subdirectory of _TMTGTBUILDROOT as follows:

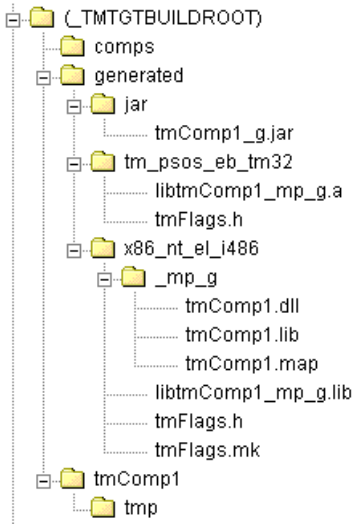


Figure 3-4: Library directory structure if _TMTGTBUILDROOT is not empty

If _TMTGTBUILDROOT is undefined or empty the library directory structure is a part of the SDE2 **comps** directory (next diagram).

The **tmp** directory is a subdirectory of the specific component's root directory. It contains the intermediate results when building is done for the component Comp1. It contains object, dependency and option files. The **generated** directory is a subdirectory of the **comps** directory.

The overall directory structure when _TMTGTBUILDROOT is undefined or empty is shown below:

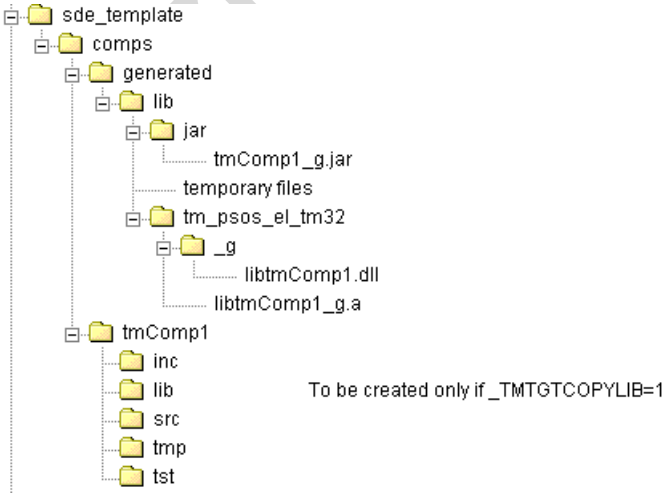


Figure 3-5: Library directory structure if _TMTGTBUILDROOT is undefined or empty

If the environment variable _TMTGTCOPYLIB is set to 1, the `lib` tree is copied to the component-specific directory (in this case to **`comps/tmComp1/lib`**).

The complete library location path is given in *Section 3.2.3, Identifying the configuration of libraries and executables* on page 28.

DLLs are treated like static libraries; they are placed in the subdirectory of the release library directory. If found in this location, a DLL is used during the link process, otherwise a static library is used. All of the operations described above for libraries are valid for DLLs too. The only differences are DLL names, they are described in *Section 3.6, Dynamic link libraries (DLLs)* on page 42.

The JAR (Java Archive Class) files are located in the **`lib/jar`** subdirectory. They are the same for all platforms, so they are platform-independent. For more about them you can read *Appendix A*.

The main purpose of this directory structure approach is that you can deliver the whole component to the customer without additional operations.

The **`lib`** directory contains a subdirectory **`tm_psos_el_tm32`**. Then directory name indicates that this subdirectory contains TriMedia32-pSOS little-endian libraries. The name of the generated library ends with _g indicating that this is a debug library. Besides the debug mode, there are also assert, retail and trace modes. See *Section 3.4.6, Debug, assert, trace and retail libraries* on page 35.

3.1.12 Executables location

The test executables (called also tests, executables) location path is given in, *Identifying the configuration of libraries and executables*.

The temporary and intermediate files for the executables build are stored in **`<_SDE_TMTGTBUILDROOT>/comps/<component>/tst/<test name>/tmp`** directory, where _SDE_TMTGTBUILDROOT is equal to _TMTGTBUILDROOT if it is defined. Otherwise it is equal to _TMROOT .

3.2 Configurations

Libraries and executables can be built for different configurations. Configurations are defined by endianness, CPU-type, operating system, and so on. The next two sections describe how to select and identify a configuration.

3.2.1 Selecting a configuration

Before invoking make or gmake for a component or executable, SDE2 must be initialized, i.e., some environment variables must be set to indicate the target build. Table 3-1 shows the four main environment variables and their possible values/combinations. The _TMTGTOSCLASS, _TMTOOLCHAIN and _TMTGTCPUCLASS variables form the so-called configuration class. As a rule, they do not contain capitals (only small letters and numbers). Each configuration class corresponds to a certain tool set. If the _TMTOOLCHAIN is undefined, a default toolchain value would be used for that particular configuration. This default toolchain value would be extracted from default_toolchain.mk file. Any new configuration defined from now onwards should contain _TMTOOLCHAIN value.

Example: The configuration class **mips_psos** corresponds to the ISI Diab Data tool set and supports the Diab data compiler.

Table 3-1: Configuration classes and their environment variables

Configuration class	General environment variables and their values
8051keil_nullos 8051 Keil compiler	**_TMTGTOS=nullos** **_TMTGTOSCLASS=nullos** **_TMTGTCPUTYPE=8051** **_TMTGTCPUCLASS=8051** **_TMBSL=** **_TMTOOLCHAIN=keil**
arm_ce WinCE compiler default toolchain - ms	**_TMTGTOS=ce300** **_TMTGTOSCLASS=ce** **_TMTGTCPUTYPE=arm720** **_TMTGTCPUCLASS=arm** **_TMBSL= or _TMBSL=_dvp1**
arm_cexec arm ads compiler default toolchain - ads	**_TMTGTOS=cexec** **_TMTGTOSCLASS=cexec** **_TMTGTCPUTYPE=arm7, arm920t, arm922t, arm940t, arm10, strongarm, arm926EJS, arm946** **_TMTGTCPUCLASS=arm** **_TMBSL=**
arm_nullos ARM-ELF-GCC for Linux default toolchain - gnu	**_TMTGTOS=nullos** **_TMTGTOSCLASS=nullos** **_TMTGTCPUTYPE=arm7, arm920t, arm940t, arm10, strongarm, arm926EJS, arm946** **_TMTGTCPUCLASS=arm** **_TMBSL= or _TMBSL=_dvp1** **_TMTOOLCHAIN=**
arm_vxworks Tornado toolchain for arm default toolchain - trnd	**_TMTGTOS=vxworks540, or vxworks** **_TMTGTOSCLASS=vxworks** **_TMTGTCPUTYPE=arm7, arm920t, arm922t, arm940t, arm10, strongarm, arm926EJS, arm946** **_TMTGTCPUCLASS=arm** **_TMBSL= or _TMBSL=_dvp1**
armads_nucleus arm ads compiler	**_TMTGTOS=nucleus** **_TMTGTOSCLASS=nucleus** **_TMTGTCPUTYPE=arm7, arm920t, arm922t, arm940t, arm10, strongarm, arm926EJS, arm946** **_TMTGTCPUCLASS=arm** **_TMBSL=** **_TMTOOLCHAIN=ads**
armads_nullos arm ads compiler	**_TMTGTOS=nullos** **_TMTGTOSCLASS=nullos** **_TMTGTCPUTYPE=arm7, arm920t, arm922t, arm940t, arm10, strongarm, arm926EJS, arm946, arm720** **_TMTGTCPUCLASS=arm** **_TMBSL= or _TMBSL=_dvp1** **_TMTOOLCHAIN=ads**

Table 3-1: Configuration classes and their environment variables <Helv9R>(Cont'd.)

Configuration class	General environment variables and their values
armads_ucos arm ads compiler	**_TMTGTOS=ucos** **_TMTGTOSCLASS=ucos** **_TMTGTCPUTYPE=arm7, arm920t, arm922t, arm940t, arm10, strongarm, arm926EJS, arm946, arm720** **_TMTGTCPUCLASS=arm** **_TMBSL= or _TMBSL=_evaluator** **_TMTOOLCHAIN=ads**
armghs_nullos arm GreenHills compiler	**_TMTGTOS=nullos** **_TMTGTOSCLASS=nullos** **_TMTGTCPUTYPE=arm7, arm920t, arm922t, arm940t, arm10, strongarm, arm926EJS, arm946, arm720** **_TMTGTCPUCLASS=arm** **_TMBSL=** **_TMTOOLCHAIN=ghs**
armghs_oscan arm GreenHills compiler for osCAN	**_TMTGTOS=nullos** **_TMTGTOSCLASS=nullos** **_TMTGTCPUTYPE=arm7, arm920t, arm922t, arm940t, arm10, strongarm, arm926EJS, arm946, arm720** **_TMTGTCPUCLASS=arm** **_TMBSL=** **_TMTOOLCHAIN=ghs**
armgnu_linux arm support for Linux target OS	**_TMTGTOS=linux** **_TMTGTOSCLASS=linux** **_TMTGTCPUTYPE=arm7, arm920t, arm922t, arm940t, arm10, strongarm, arm926EJS, arm946, arm966es, arm968es, arm1020e, arm1022e, arm1026ejs, arm1156t2fs, arm1156t2s, arm1176jtzs, arm1176JTZS** **_TMTGTCPUCLASS=arm** **_TMBSL=** **_TMTOOLCHAIN=gnu**
armrvds_nullos arm RealView compiler	**_TMTGTOS=nullos** **_TMTGTOSCLASS=nullos** **_TMTGTCPUTYPE=arm7, arm920t, arm922t, arm940t, arm10, strongarm, arm926EJS, arm946, arm1176jtzs, arm1176JTZS** **_TMTGTCPUCLASS=arm** **_TMBSL=** **_TMTOOLCHAIN=rvds**
armrvds_ucos arm RealView compiler This configuration changes are specific to (MST Nexperia Mobile Multimedia BL Cordless & Imaging BU Mobile Communications NXP Semiconductors Sophia, France)	**_TMTGTOS=ucos** **_TMTGTOSCLASS=ucos** **_TMTGTCPUTYPE=arm7, arm920t, arm922t, arm940t, arm10, strongarm, arm926EJS, arm946, , arm966es, arm968es, arm1020e, arm1022e, arm1026ejs, arm1156t2fs, arm1156t2s, arm1176jtzs, arm1176JTZS** **_TMTGTCPUCLASS=arm** **_TMBSL=_PNX4002 or _osaltu** **_TMTOOLCHAIN=rvds**
hp_nullos standard GCC for HP-UX default toolchain - gnu	**_TMTGTOS=nullos** **_TMTGTOSCLASS=nullos** **_TMTGTCPUTYPE=hp** **_TMTGTCPUCLASS=hp** **_TMBSL= or _TMBSL=_dvp1**

Table 3-1: Configuration classes and their environment variables <Helv9R>(Cont'd.)

Configuration class	General environment variables and their values
mips_ce WinCE compiler default toolchain - ms	**_TMTGTOS=ce300** **_TMTGTOSCLASS=ce** **_TMTGTCPUTYPE=r3940, r4300, or r4640, or r4450** **_TMTGTCPUCLASS=mips** **_TMBSL= or _TMBSL=_dvp1**
mips_nullos default toolchain - gnu	**_TMTGTOS=nullos** **_TMTGTOSCLASS=nullos** **_TMTGTCPUTYPE=r1910, r3940, r4300, r4640 or r4450** **_TMTGTCPUCLASS=mips** **_TMBSL=**
mips_psos (ISI Diab Data compiler) default toolchain - diab	**_TMTGTOS=psos250** **_TMTGTOSCLASS=psos** **_TMTGTCPUTYPE= r4300, r4640, 4450, r3940, r1910, r4450, 4kec, r4000 or mips32** **_TMTGTCPUCLASS=mips** **_TMBSL=_p4032 or _TMBSL=_dvp1**
mips_vxworks Tornado toolchain for mips default toolchain - trnd	**_TMTGTOS=vxworks540, or vxworks** **_TMTGTOSCLASS=vxworks** **_TMTGTCPUTYPE=r3940, r4300, or r4640, or r4450** **_TMTGTCPUCLASS=mips** **_TMBSL= or _TMBSL=_dvp1**
mipsghs_integrity Green Hills compiler toolchain	**_TMTGTOS=integrity** **_TMTGTOSCLASS=integrity** **_TMTGTCPUTYPE= 4kec** **_TMTGTCPUCLASS=mips** **_TMBSL=simr5000**
mipsghs_nullos Green Hills compiler toolchain	**_TMTGTOS=nullos** **_TMTGTOSCLASS=nullos** **_TMTGTCPUTYPE= 4kec, 24kc, 24kf, 24kec, 24kef** **_TMTGTCPUCLASS=mips** **_TMBSL=generic**
mipsgnu_ecos mips support for eCOS OS	**_TMTGTOS=ecos** **_TMTGTOSCLASS=ecos** **_TMTGTCPUTYPE=4kec** **_TMTGTCPUCLASS=mips** **_TMBSL=** **_TMTOOLCHAIN=gnu**
mipsgnu_linux mips support for Linux target OS	**_TMTGTOS=linux** **_TMTGTOSCLASS=linux** **_TMTGTCPUTYPE=r3940, r4300, or r4640, or r4450** **_TMTGTCPUCLASS=mips** **_TMBSL=** **_TMTOOLCHAIN=gnu**
real_mtos Real DSP Compiler default toolchain - ace	**_TMTGTOS=mtos** **_TMTGTOSCLASS=mtos** **_TMTGTCPUTYPE=rd24120** **_TMTGTCPUCLASS=real** **_TMBSL= or _TMBSL=_dvp1**

Table 3-1: Configuration classes and their environment variables <Helv9R>(Cont'd.)

Configuration class	General environment variables and their values
real_nullos Real DSP Compiler default toolchain - ace	**_TMTGTOS=nullos** **_TMTGTOSCLASS=nullos** **_TMTGTCPUTYPE=rd24120** **_TMTGTCPUCLASS=real** **_TMBSL= or _TMBSL=_dvp1**
realsat_nullos Saturn DSP compiler toolchain	**_TMTGTOS=nullos** **_TMTGTOSCLASS=nullos** **_TMTGTCPUTYPE=rd16023, rd16024** **_TMTGTCPUCLASS=real** **_TMBSL=** **_TMTOOLCHAIN=sat**
tm_psos Trimedia TCS compiler toolchain default toolchain - tcs	**_TMTGTOS=psos200, psos250 or psostm200** **_TMTGTOSCLASS=psos** **_TMTGTCPUTYPE=tm32, tm3260, tm1100, or tm1300** **_TMTGTCPUCLASS=tm** **_TMBSL= or _TMBSL=_dvp1**
tmtcs_nullos Trimedia TCS compiler toolchain default toolchain - tcs	**_TMTGTOS=nullos** **_TMTGTOSCLASS=nullos** **_TMTGTCPUTYPE=tm32, tm3260, tm3270, tm3271, tm1100, or tm1300** **_TMTGTCPUCLASS=tm** **_TMBSL= or _TMBSL=_dvp1**
x86_ce WinCE compiler default toolchain - ms	**_TMTGTOS=ce300** **_TMTGTOSCLASS=ce** **_TMTGTCPUTYPE=i486** **_TMTGTCPUCLASS=x86** **_TMTGTENDIAN=el** **_TMBSL= or _TMBSL=_dvp1**
x86_nt Micro soft Developer Studio default toolchain - ms	**_TMTGTOS=nt4** **_TMTGTOSCLASS=nt** **_TMTGTCPUTYPE=i486** **_TMTGTCPUCLASS=x86** **_TMBSL= or _TMBSL=_dvp1**
x86_vxworks Tornado toolchain for x86 default toolchain - trnd	**_TMTGTOS=vxworks540, or vxworks** **_TMTGTOSCLASS=vxworks** **_TMTGTCPUTYPE=i486** **_TMTGTCPUCLASS=x86** **_TMBSL= or _TMBSL=_dvp1**

Table 3-1: Configuration classes and their environment variables <Helv9R>(Cont'd.)

Configuration class	General environment variables and their values
x86ddk_nt Microsoft DDK (Driver Development Kit)	**_TMTGTOS=nt4** **_TMTGTOSCLASS=nt** **_TMTGTCPUTYPE=i486** **_TMTGTCPUCLASS=x86** **_TMBSL=** **_TMTOOLCHAIN=ddk**
x86gnu_linux x86 support for Linux target OS	**_TMTGTOS=linux** **_TMTGTOSCLASS=linux** **_TMTGTCPUTYPE=i486** **_TMTGTCPUCLASS=x86** **_TMBSL=**
x86gnu_nullos default toolchain - gnu	**_TMTGTOS=nullos** **_TMTGTOSCLASS=nullos** **_TMTGTCPUTYPE=i486** **_TMTGTCPUCLASS=x86** **_TMBSL=**

SDE2 (2.1 onwards) also supports the following configurations for **System-C** support

Configuration Class	General environment variables and their values
hpncsc_nullos Cadence-NcSc compiler toolchain for NxBuilder Support	**_TMTGTOS=nullos** **_TMTGTOSCLASS=nullos** **_TMTGTCPUTYPE=hp** **_TMTGTCPUCLASS=hp** **_TMBSL=** _TMTOOLCHAIN=ncsc
x86ncsc_nullos Cadence-NcSc compiler toolchain for Linux	**_TMTGTOS=nullos** **_TMTGTOSCLASS=nullos** **_TMTGTCPUTYPE=i486** **_TMTGTCPUCLASS=x86** **_TMBSL=** _TMTOOLCHAIN=ncsc
x86osci_nullos OSCI compiler toolchain for Linux	**_TMTGTOS=nullos** **_TMTGTOSCLASS=nullos** **_TMTGTCPUTYPE=i486** **_TMTGTCPUCLASS=x86** **_TMBSL=** _TMTOOLCHAIN=osci
x86osci_nt OSCI compiler toolchain on Windows	**_TMTGTOS=nt4** **_TMTGTOSCLASS=nt** **_TMTGTCPUTYPE=i486** **_TMTGTCPUCLASS=x86** **_TMBSL=** _TMTOOLCHAIN=osci

Please refer to *Chapter 4* for more information on System-C support in SDE2

3.2.2 Configuration check

SDE2 would check for the wrong combination of CPUCLASS and CPUTYPE, OSCLASS and OSTYPE set in the environment. Eg: If you set CPUCLASS and OSCLASS as **x86** and **nt** respectively, then set CPUTYPE as **hp**. SDE2 would throw an error message notifying that **hp** is not of CPUCLASS **x86**. This is done by the **sde/config_check.mk** included in **sde/environment.mk** file

Per configuration class, some extra environment variables have to be set as listed in the following table, Table 3-2.

Note: Do not use spaces in the environment names on a WinNT host. Use DOS-compliant names!

Table 3-2: Additional environment variables per configuration class

Configuration class	Environment variables with explanations
8051keil_nullos	**KEILTOOLSET = <drive>:/keil**
arm_cexec	**_ARMADSTOOLCHAIN=<drive>:/ARM/ADSv1_1/include**
arm_nullos	No configuration specific variables need to be set for **arm_nullos**
arm_vxworks	**WIND_HOST_TYPE=x86-win32** **WIND_BASE=E:\Tornado** **PATH=%WIND_BASE%\host\% WIND_HOST_TYPE%\bin;%PATH%**
armads_nullos, armads_nucleus armads_ucos, armrvds_nullos	**_ARMADSTOOLCHAIN=<drive>:/ARM/ADSv1_1/include** **GCPP=gcc (for assembler source code)**
armgnu_linux, mipsgnu_linux, x86gnu_linux	**GCC_BASE=<path to installation of GCC>** **GCC_PREFIX=<Cross compiler prefix>** **GCC_VERSION=<GCC version being used>**
armghs_nullos	**GHS=<drive>:/GHS/Arm401**
armrvds_ucos	No configuration specific variables need to be set for **armrvds_ucos**
hp_nullos (Standard GCC for HP-UX)	No configuration specific variables need to be set for **hp_nullos**
mips_psos (ISI Diab Data compiler)	**ISIMIP**: Location of **`isimip`** directory **LM_LICENSE_FILE**: Location of license file **DIABLIB=%ISIMIP%/diab/4.3p5** **PSS_ROOT=%ISIMIP%/pssmip.250** **PATH=%ISIMIP%\licenses\bin\win32;** **%ISIMIP%\diab\4.3p5\win32\bin;%PATH%** **PSS_BSP=%PSS_ROOT%/bsps/p4032** **DFP=H or DFP=S** （for executables only, default **DFP=S**)
mips_vxworks	**WIND_HOST_TYPE=x86-win32** **WIND_BASE=E:\Tornado** **PATH=%WIND_BASE%\host\% WIND_HOST_TYPE%\bin;%PATH%**
mipsghs_integrity	GHS_HOME=<drive>:/GHS/int504 _TMBSL=simr5000 (_TMBSL is a must for this configuration)
real_mtos	**COMPROOT=/real/v010100/RD24120** **RCC_OPTIONS=**
real_nullos	**COMPROOT=/real/v010100/RD24120** **RCC_OPTIONS=**
realsat_nullos	No configuration specific variables need to be set for **realsat_nullos**

Table 3-2: Additional environment variables per configuration class <Helv9R>(Cont'd.)

Configuration class	Environment variables with explanations
tm_psos	**TCS**: Location of TriMedia directory **_TMTCSHOST**: TriMedia host; Possible values: **WinNT**, **nohost**, **tmsim** SDE2 supports both **TCS 2.2** and **TCS 4.2.** Point **TCS** to **TCS 4.2** installation.
All WinCE configurations: **arm_ce, mips_ce, x86_ce** (WinCE compiler) **Note:** Italicized values are available with the Microsoft WinCE Platform Builder, but they are unsupported in SDE2. **Note:** See Section 3.2.2.1, *Compilation in a WinCE 3.00 Environment* for more information about the WinCE 3.0 environment.	**_FLATRELEASEDIR=<drive>:/WINCE300/release** **_WINCEROOT=<drive>:/WINCE300** **CEPBDIR=E:\Progra~1\Window~1\3.00\CEPB\Bin** **_TGTCPU=R3000 R4100 R4111 R4200 R4300 *PPC821 SH3 SH4* i486 *SA1100 ARM920* ARM720 *PPC403 IDT32364 CEF*** **_TGTCPUTYPE=ARM MIPS *PPC SHx* x86 THUMB *CEF*** **_TGTOS=NT NTANSI CE** **_TGTPLAT=NOPLAT DESKTOP DESKTOP_SDK ODO ...** **_TGTPROJ=MINSHELL MAXSHELL ...** **_PROJECTROOT=E:\WINCE300\public\<_TGTPROJ>** **_TARGETPLATROOT=E:\WINCE300\platform\ <_TGTPLAT>** **_OEMINCPATH=<list of directories>**
x86_nt (Developer Studio)	**VCC**: Location of Developer Studio binaries. **VCC**-specific settings by calling: **%VCC%\bin\vcvars32.bat**
x86ddk_nt	DDK_HOME=<drive>:/WINDDK/3790
x86gnu_nullos	No configuration specific variables need to be set for **x86gnu_nullos**

Table 3-3: Additional environment variables for System C components

Table 3-4:

Configuration class	Environment variables with explanations
hpncsc_nullos	No configuration specific variables need to be set for **hpncsc_nullos**
x86ncsc_nullos	**_TMPSE = <location of Philips SystemC Environment> optional**
x86osci_nullos	**_TMPSE = <location of Philips SystemC Environment> optional**

Finally, there are some generic variables that need to be set for each configuration class, see Table 3-5.

Table 3-5: The generic environment variables

Environment variable	Description	Possible values
_TMSITE	Site being worked on	Local value like **ehvblv**, **hbgslh**, **svlssg**, **svldvi**, **blrsdm**
_TMROOT	Location of your SDE2 structure	Any absolute path with forward slashes.
UNAME	Specifies the host platform	**hpux**, **linux** or **cygwin** (for WinNT)
_TMTGTBUILDROOT	Location of product files (generated files)	If not empty, any absolute path with forward slashes. If empty, **comps** directory.
_TMECHO	Show extra info during **make** (or **gmake**)	Not empty or empty/undefined
_TMTGTCOPYLIB	Copy the lib directory tree to a component-specific directory	1 (copy) or other (do nothing)

Table 3-5: The generic environment variables <Helv9R>(Cont'd.)

Environment variable	Description	Possible values
_TMTGTCOPYOBJ	Copy the generated object files to a component-specific directory	Space-separated list of objects without their extension, for example, **tmComp1 tmComp2**
_TMBSL	Specifies the link rules for building an image. You can make your custom file **maketarget<_bsl>.mk** and set **_TMBSL=<_bsl>**.	Can be a user-defined string specifying the target platform; shows the default values per configuration class.
_TMTGTENDIAN	Endianness. Set for each configuration class. It plays significant role only for TriMedia (**tm_psos**). Different libraries are generated.	**eb** or **el** (Obsolete values **be** and **le** are still accepted in this release.)
_TMTGTREL	Release type	**debug**, **assert**, **trace** or **retail** (Obsolete values **dbg** and **ret** are still accepted in this release.)
_TMLINKTYPE	Link type (static/dynamic- DLL), required for executables. Only **. It is currenly used only for tm_psos**	**static** or **dynamic**
_TMDIVERSITY	User-defined string for diversity, for example, **_mp_flo_**	Normally empty, see *Section 3.5, Diversities* on page 36. If non empty, different diversities are separated with an underscore.
PATH	When using the SDE2 with WinNT, the path variable is extended by SDE2 with the Cygwin tool set (delivered with the SDE2). Use backlashes.	**PATH%=_TMROOT\sde\cygwin;%PATH%**
_TMNODEPENDENCIES	Dependency checking can be switched off by setting to 1. No dependency checking results in faster compilation times.	1 (switch checking off) and other (do dependency checking).
_TMNESTEDINCLUDE	Recursive closure of Requires section can be done. No recursive closure of requires section has less includle paths.	null (novalue switch off) no recursive closure of requires and 1 or any value (any value swithc on) recursive closure of requires.
_TMTGTWARNINGS	Level of warnings. Level 0 corresponds to (almost) no warnings and level 3 to (almost) all warnings.	0, 1, 2, 3 or undefined

Note: Most environment variables are directly used by SDE2 and require forward slashes (For Windows only). The **PATH** variable is not directly used by SDE2, but is required for a WinNT host and requires backlashes.

Compilation of components for a certain configuration is done by opening a DOS or UNIX shell, executing the platform-specific batch file (usually located in **project/sites/<your site>** directory), going to the component's directory and typing **gmake**.

Choosing a configuration means setting the environment variables of Table 3-1, the relevant environment variables of Table 3-2 and Table 3-5. Setting these variables is typically done by a batch file such as **tm_psos_debug_static_el_tm32_winnt_default.bat,** which is in the directory **project/sites/blrsdm** or **project/sites/<your site>**. Below, such a file for a **tm_psos** configuration class is displayed:

```
set _TMSITE=blrsdm
set _TMROOT=C:/ccm_wa/ssgmoreuse/sde-kazakov/sde/sde_template
set UNAME=cygwin
set _TMTGTBUILDROOT=c:/b_result_1812
set PATH=%_TMROOT%\sde\cygwin;%PATH%
set _TMECHO=
rem host and trimedia psos specific settings
set TCS=c:/trimedia
rem trimedia psos specific settings
set _TMTGTOS=psos250
set _TMTGTOSCLASS=psos
set _TMTGTCPUTYPE=tm32
set _TMTGTCPUCLASS=tm
set _TMTCSHOST=WinNT
rem flavor settings
set _TMBSL=
set _TMTGTENDIAN=el
set _TMTGTREL=debug
set _TMLINKTYPE=static
rem optional flavor setting
set _TMDIVERSITY=
```

Now we will discuss some specific issues for certain configurations.

3.2.2.1 Compilation in a WinCE 3.00 Environment

In this section it is described how to use SDE2 in combination with the WinCE 3.00 environment. This is not an easy matter and this manual does not give a complete overview of what is to be done, but only some experiences of SDE2 team that might be useful to the reader.

It is important that the appropriate environmental settings are set correctly. Working with Microsoft Platform Builder may be rather difficult.

It is assumed that Microsoft Platform Builder 3.00 has been installed. Furthermore it is assumed that you have a platform (CEPC or ODO in standard installation) and a Public (Project) directory (in the following example **MaxAll** and **MinShell** will be used).

The first step is to build the WinCE 3.00 environment. This can be done in a DOS window by entering a command like

CMD.EXE /k E:\WINCE300\public\common\oak\misc\wince.bat ARM ARM720 CE MAXALL ODO
or

CMD.EXE /k E:\WINCE300\public\common\oak\misc\wince.bat MIPS R3000 CE MAXALL ODO

or

CMD.EXE /k E:\WINCE300\public\common\oak\misc\wince.bat x86 i486 CE MAXALL CEPC

The parameters after **wince.bat** are stored in the environmental variables **_TGTCPUTYPE**, **_TGTCPU**, **_TGTOS**, **_TGTPROJ**, **_TGTPLAT**. Also used is the value of the variable **WINCEDEBUG** that defines the compilation mode. The compilation mode in WinCE can be retail or debug[3]and it is responsible for building libraries for one of these modes.

The next step is to execute **buildrel** and **blddemo** to build libraries and configuration files for the chosen configuration. The libraries are placed in the directory **<_WINCEROOT>/Public/<_TGTPROJ>/cesysgen**.

The value of the environmental variable **_OEMINCPATH** is obtained from the file **<_TARGETPLATROOT>/sources.gen** that is generated via Platform Builder.

The compilation for the DVP1 software is done using the DVP1 platform instead of ODO. Other custom platforms can also be used.

Note: Make sure that your WinCE directories are placed before the Microsoft VCC directories in the PATH variable.

The SDE2 team was not able to build the configuration for the **ARM920** processor.

For the ARM processor, only ARM720 with THUMB mode is supported.

Note: If you compile both for WinCE and **x86_nt**, the compiler may get confused picking the wrong header files; it is better to put WInCE setting before **x86_nt**, but we would advise you not to mix these two configurations in one shell environment.

3.2.2.2 Compilation in a R.E.A.L. environment

SDE2 supports R.E.A.L. environment for the toolset developed by ACE Associated Compiler Experts B.V. This toolset has very severe restrictions of the C syntax/compiler/linker and therefore it is unlikely components developed for other platform to be reused for R.E.A.L. However, the components developed for R.E.A.L. can be reused on another platform. This compiler is supported for mtos OS and for nullos OS (does not use OS-specific features).

3.2.2.3 Compilation with VxWorks OS

The **arm_vxworks** configuration is not tested for some processors. There are known issues with VxWorks in this release. One is support for Thumb mode (16-bit mode). For the current version of Tornado (2.1) it is not possible to mix Thumb and ARM mode according to the documentation. Our understanding of how to handle this is that this is component diversity (see *Section 3.5, Diversities* on page 36 for more information). We suggest the following for your component's **diversity.mk**:

```
ifeq ($(_TMTGTCPUCLASS)),arm)
ifeq ($(findstring _thumb_,$(_TMDIVERSITY)),_thumb)
_<Your component name>_DIVERSITY=_thumb
endif
endif
```

and in your makefile:

```
include diversity.mk
LIB_SUFFIX := _<Your component name>_DIVERSITY
```

3. There is no assert mode in WinCe 3.00.

The following defines were made:

For 32-bit mode (this is determined if **LIB_SUFFIX** does not contain **_thumb_**:

-DCPU=ARMARCH4 -march=armv4

For (16-bit) Thumb mode (this is determined if **LIB_SUFFIX** contains **_thumb_**:

-DCPU=ARMARCH4_T -march=armv4t -mthumb -mthumb-interwork

For ARM7 processors (we assume **arm720t** is used, override **ARM_CPU** if not):

-DARMMMU=ARMMMU_720T -DARMCACHE=ARMCACHE_720T

For ARM920T processors:

-DARMMMU=ARMMMU_920T -DARMCACHE=ARMCACHE_920T

For ARM940T processors:

-DARMMMU=ARMMMU_940T -DARMCACHE=ARMCACHE_940T

If you use VxWorks, be aware that WindRiver (producer of VxWorks) changed the standard gmake utilities, so they are not standard anymore. Therefore, for the compilation of the dependencies we are using a C compilation utility from WindRiver (included in the installation, it depends on the platform) instead of the standard gmake one.

We do not use standard libraries (such as **libgcc.a**) by default. The VxWorks linker does and to prevent this, we use the option **-nostdlib**.

Be aware also, that **arm_vxworks** and **mips_vxworks** platforms are not consistent with each other. This is due to the fact that the current working projects for these configurations are very different. However, in a future release we are going to make them more consistent.

3.2.2.4 Building executables for mips_psos

There is one specific issue for building executables for **mips_psos**. In order to do this, we need to build a number of object files in advance. They depend on the psos **sys_conf.h** file. This is a time-costly operation and therefore, we build these object files once. If you use the global **sys_conf.h** file, these object files are located in the **`generated`** directory and its subdirectories (i.e., it is global). The file **libstart*.a** is generated from the object files and it is used during the linking. If you have a local **sys_conf.h** file (usually in the **`inc`** directory), SDE2 generates all object files and **libstart_local*.a** file. This file is always generated, because it is placed in a global directory. The object files are generated once and they are placed in a component-specific directory.

3.2.3 Identifying the configuration of libraries and executables

The location and the name of a library identify the configuration a library has been built for. Below, the location/name of a library is shown in the case where **_TMTGTBUILDROOT** is not empty:

`<_TMTGTBUILDROOT>/comps/generated/lib/<_TMTGTCPUCLASS><_TMTOOLCHAIN>_<_TMTGTOSCLASS>_<_TMTGTENDIAN>_<_TMTGTCPUTYPE>/lib<NAME><LIB_SUFFIX><REL_SUFFIX>.<EXT>`

For example,

build/comps/generated/lib/tm_psos_el_tm32/libtmRealFloat_g.a

Here _TMTGTBUILDROOT, _TMTGTCPUCLASS, _TMTGTOSCLASS, _TMTGTENDIAN, and _TMTGTCPUTYPE are the environment variables as explained in Section 3.2.1, *Selecting a configuration*, and NAME is the name of the component with the prefix tm. The NAME is extracted from the DIR_LOCAL variable, which must be present in the beginning of each makefile, for example,

DIR_LOCAL = comps/tmComp1

In this case NAME is tmComp1.

For libraries, EXT is generated by the SDE2 and equals `lib` for x86_nt, mips_ce, x86_ce, arm_ce, and a for the rest. The value of REL_SUFFIX is usually[4], and the value of REL_COMPS_SUFFIX is always equal to _g when _TMTGTREL is debug, _a when _TMTGTREL is assert, "" (empty string) when _TMTGTREL is retail and _t when _TMTGTREL is trace. The value of LIB_SUFFIX can be defined in the component makefile. As a rule, it depends on the diversity and contains different components' diversities (see Section 3.5, *Diversities* for more information) separated with underscores. For example,

LIB_SUFFIX = _mp_flo

The location of the DLL files is in the subdirectory of the corresponding static library directory. The name of this subdirectory depends on _TMDIVERSITY and REL_COMPS_SUFFIX. For more information see Section 3.6, *Dynamic link libraries (DLLs)*.

JAR files are not platform-dependent. They are placed in the jar subdirectory of lib, for example,

build/comps/generated/lib/jar/tmComp12_g.jar

Executables are stored in the following way:

<_TMTGTBUILDROOT>/comps/<component>/tst/Tst1/bin/<_TMTGTCPUCLASS><_TMTOOLCHIAN>_<_TMTGTOSCLASS>_<_TMLINKTYPE>_<_TMTGTENDIAN>_<_TMTGTCPUTYPE><REL_SUFFIX>_<_TMTCSHOST><_TMBSL><_TMDIVERSITY>/<TARGET>.<EXT>

Applications are stored in the `<_TMTGTBUILDROOT>/apps` directory.

When _TMTGTCPUCLASS is not equal to tm then _<_TMTCSHOST> is not part of the path name. TARGET is the name of the executable, and EXT is either exe (nt, ce) or out (tm, mips, hp).

For example,

build1/comps/tmComp1/tst/Tst1/bin/tm_psos_static_el_tm32_WinNT_default/test.out

3.3 Standard precompile flags and tmFlags.h file

A lot of C/C++ code uses precompile flags to do conditional compilation of code fragments. An example is shown below:

```
#ifdef NDEBUG
printf("Retail mode\n");
#else
printf("Debug mode\n");
#endif
```

4. For exceptions read Section 3.13.4.2.

SDE2 generates a header file **tmFlags.h** that contains some constants. Together with it, SDE2 generates the makefile **tmFlags.mk**, with the same logical contents as **tmFlags.h**, and the file **tmFlags.cfg** with some version settings. There is one **tmFlags.h** file generated for each library directory (example **tm_psos_el_tm32**). This is because this file contains in addition to definitions, other choices. For instance the file defines all possible endiannesses (**TMFL_ENDIAN_BIG**, **TMFL_ENDIAN_LITTLE**) and the choice **TMFL_ENDIAN** which is then either **TMFL_ENDIAN_BIG** or **TMFL_ENDIAN_LITTLE**. All defines related to OS, CPU, and endianness are made by SDE2, and the libraries compiled for different combinations cannot be linked together. Next to the predefined choices, the file contains standard diversity constants, like **TMFL_REL_DEBUG**, **TMFL_REL_ASSERT**, **TMFL_REL_RETAIL** and **TMFL_REL_TRACE**. The choice is not made in this **tmFlags.h** file because retail and debug libraries can be linked together (they are binary compatible). There are more of these predefined but not selected defines, like **TMFL_SCOPE_XX** for processor scope and the file can be extended for all globally useful diversities.

The constants in the file are also available in the make process. So besides an **ifdef** in the code, **if** statements in makefiles are allowed based on these flags.

Example of source code:

```
#if (TMFL_ENDIAN==TMFL_ENDIAN_BIG)
        # do some big endian specific stuff
#else
        # do some little endian specific stuff
#endif
```

Example of makefile:

```
ifeq ($(TMFL_ENDIAN),$(TMFL_ENDIAN_BIG))
        C_SOURCES += my_big_endian_convertor.c
else
        C_SOURCES += my_little_endian_convertor.c
endif
```

Table 3-6: #define variables that can be queried directly

#define that can be queried	Possible values
TMFL_CPU_IS_8051	0 or 1
TMFL_CPU_IS_ARM	0 or 1
TMFL_CPU_IS_HP	0 or 1
TMFL_CPU_IS_MIPS	0 or 1
TMFL_CPU_IS_REAL	0 or 1
TMFL_CPU_IS_TM	0 or 1
TMFL_CPU_IS_X86	0 or 1
TMFL_OS_IS_BTM	0 or 1
TMFL_OS_IS_CE	0 or 1
TMFL_OS_IS_CEXEC	0 or 1
TMFL_OS_IS_ECOS	0 or 1

Table 3-6: #define variables that can be queried directly <Helv9R>(Cont'd.)

#define that can be queried	Possible values
TMFL_OS_IS_INTEGRITY	0 or 1
TMFL_OS_IS_LINUX	0 or 1
TMFL_OS_IS_MTOS	0 or 1
TMFL_OS_IS_NT	0 or 1
TMFL_OS_IS_NUCLEUS	0 or 1
TMFL_OS_IS_NULLOS	0 or 1
TMFL_OS_IS_PSOS	0 or 1
TMFL_OS_IS_UCOS	0 or 1
TMFL_OS_IS_VXWORKS	0 or 1

Table 3-6: #define variables that can be queried directly <Helv9R>(Cont'd.)

#define that can be queried	Possible values
TMFL_ENDIAN	TMFL_ENDIAN_BIG TMFL_ENDIAN_LITTLE
TMFL_CPU	TMFL_CPU_4KEC TMFL_CPU_8051 TMFL_CPU_ARM10 TMFL_CPU_ARM1020E TMFL_CPU_ARM1022E TMFL_CPU_ARM1026EJS TMFL_CPU_ARM1156T2FS TMFL_CPU_ARM1156TFS TMFL_CPU_ARM1176JTZS TMFL_CPU_ARM720 TMFL_CPU_ARM922T TMFL_CPU_ARM926EJS TMFL_CPU_ARM946 TMFL_CPU_ARM966ES TMFL_CPU_ARM968ES TMFL_CPU_MIPS32 TMFL_CPU_R1910 TMFL_CPU_R3900 TMFL_CPU_R4000 TMFL_CPU_R4450 TMFL_CPU_RD16023 TMFL_CPU_RD16024 TMFL_CPU_RD24120 TMFL_CPU_STRONGARM TMFL_CPU_TM3260
TMFL_OS	TMFL_OS_BTM TMFL_OS_CE TMFL_OS_CE212 TMFL_OS_CE300 TMFL_OS_CEXEC TMFL_OS_ECOS TMFL_OS_INTEGRITY TMFL_OS_LINUX TMFL_OS_MTOS TMFL_OS_NT TMFL_OS_NT4 TMFL_OS_NUCLEUS TMFL_OS_NULLOS TMFL_OS_PSOS TMFL_OS_PSOS200 TMFL_OS_PSOS250 TMFL_OS_UCOS TMFL_OS_VXWORKS

Here NULLOS means no OS specific settings required, i.e., UNIX, Linux or Solaris. NT is not NULLOS because some extra settings must be made for SDE2.

The 0 value corresponds to false and 1 corresponds to true. Besides these #defines some definitions are supplied to the compiler by SDE2 (by using the –D option). Table 3-7 contains the variables and their possible values. The possible values are specified in tmFlags.h.

Table 3-7: Variables set by SDE2 for the compiler

Variable that can be queried in the source code	Possible values	Explanation
TMFL_REL	TMFL_REL_RETAIL TMFL_REL_ASSERT TMFL_REL_DEBUG TMFL_REL_TRACE	Release mode
NDEBUG		Defined for release mode.

3.4 Compile and link options

In this section the compile and link options that can be set in SDE2 are described. These options are divided into project-wide compile options, component-specific compile options, file-specific compile options and link options. They are described in the following four sections. These options are applicable for C, C++, JAVA and Asembly language programming.

3.4.1 Project-wide compile options

Project-wide compile options can be fed to the build process by filling the makefile variable _SDE_EXTRA_CFLAGS (or _SDE_EXTRA_CXXFLAGS for C++ files). This variable should not contain options other than the -D options, because only the –D, –U or –I options are interpreted in the same way by all C/C++ compilers. This variable is added before the LOCAL_CFLAGS (LOCAL_CXXFLAGS) variable at the compilers settings. For example,

_SDE_EXTRA_CFLAGS = -DLEGACY

3.4.2 Component-specific compile options

Some software components require extra compile flags during compilation. C and C++ compilers support this by allowing users to supply a –D<compile flag> to the options list of the compiler. SDE2 supports this by adding the following lines to the makefile of your component or executable:

```
LOCAL_CFLAGS      = <your component's C options>
LOCAL_CXXFLAGS    = <your component's C++ options>
```

Example of a C-component that requires the options -DENABLE_GRAPHICS and DENABLE_TCPIP:

```
LOCAL_CFLAGS      = -DENABLE_GRAPHICS -DENABLE_TCPIP
```

Note that the compile flags start with -D. The content of the LOCAL_CLFLAGS and LOCAL_CXXFLAGS is literally copied to the command line of the compilation process of the source files specified in the makefile. The local flags are placed before _TMTGTCOPTS (_TMTGTCXXOPTS) in the supplied order to the compiler. In this way they can overrule existing options.

Specific compilation options can be added to the environment by putting these options in _TMTGTCOPTS (_TMTGTCXXOPTS). So, the order of all compilation options for C files is

```
<SDE2 options><_SDE_EXTRA_CFLAGS><LOCAL_CLFLAGS>
<_TMTGTCOPTS><TARGET_CFLAGS>
```

Some components require the preprocessed files. The preprocess compilation process can be started by setting the environment variable **_TMTGTCPP**.

3.4.3 File-specific compile options

Some components require that some of the source files of the component be compiled with extra compile options.

The makefile of a component can indicate this by putting the following line(s) at the tail of the makefile:

$(DIR_INTERM)/<source file>.$(_SDE_O) : TARGET_CFLAGS+=<compile option>

Here **_SDE_O** is the name of the object extension (**.o**, **.obj**).

TARGET_CFLAGS is only added to the compilation rules for that specific object. Below is an example of a part of a makefile of the component **tmComp4** with three source files, two of which have special compilation options:

$(DIR_INTERM)/src/tmComp4int1.$(_SDE_O) : TARGET_CFLAGS+=-DTMCOMP4INT1
$(DIR_INTERM)/src/tmComp4int2.$(_SDE_O) : TARGET_CFLAGS+=-DTMCOMP4INT2

Note: There no **TARGET_CXXFLAGS**, use **TARGET_CFLAGS** for C++ compilation.

Also see **`comps/tmComp4/makefile`** of the SDE2 distribution.

Note that **tmComp4** does not compile under UNIX with gmake version 7.75 or lower.

3.4.4 Link options

Some libraries require additional link options. The following line in the library makefile specifies these link options:

LOCAL_LBFLAGS_POST_LIB = <local library link options>

Some test executables require additional link options. The following line in the executable makefile specifies these link options:

LOCAL_LDFLAGS = <local executable link options>

3.4.5 Include directories

The order of include directories is:

- **DIR_LOCAL** – can be defined in the makefiles
- Local **`inc`**, **`src`**, **`cfg`** directories
- SDE2 common **`inc`** directory
- SDE2 common **`inc/cfg`** directory for **tmSysCfg.h** file
- SDE_IN_SDE **`inc`** directory (All of them) - Refer Section 3.8
- SDE2 **`inc/<_TMTGTCPUCLASS>_<_TMTGTOSCLASS>`**
- Release directory (for **tmFlags.h**)

- All `inc` directories of the required components and interfaces
- LOCAL_INCLUDES – can be defined in the makefiles
- Platform-specific include directories
- _TMTGTINCLUDES – can be defined in the makefiles

Usually DIR_LOCAL is used to define other directories within the component, for example, `src/x86_nt`.

LOCAL_INCLUDES is used to overwrite platform-specific directories.

_TMTGTINCLUDES is used to add other include directories, not overwriting platform-specific ones.

The directory `inc/<_TMTGTCPUCLASS>_<_TMTGTOSCLASS>` is used only from an executable for some legacy sources. Typically it contains sys_conf.h file for some legacy application (executable) sources.

3.4.6 Debug, assert, trace and retail libraries

Libraries can be built in four ways: in debug, retail, trace or assert mode, based on the value of the environment variable _TMTGTREL. Table 3-8 lists the characteristics.

Table 3-8: The 3 release modes and their characteristics

Release mode	Value of the _TMTGTREL environment variable	Precompile flag	Library suffix	Remarks
debug	debug	TMFL_REL== TMFL_REL_DEBUG	_g	Library includes debug information
assert	assert	TMFL_REL== TMFL_REL_ASSERT	_a	No debug information, assert statement present
retail	retail	TMFL_REL== TMFL_REL_RETAIL -DNDEBUG		No debug information,empty assert statement
trace	trace	TMFL_REL== TMFL_REL_TRACE -DNDEBUG	_t	Trace informationt

Suppose there are two components (libraries) for a tm_psos little-endian configuration and the two components have been built for all three release modes. After building, there is a `comps/generated/lib/tm_psos_el_tm32` directory containing the following files:

tmFlags.h
tmComp1.a
tmComp1_a.a
tmComp1_g.a
tmComp1_t.a
tmComp2.a
tmComp2_a.a
tmComp2_g.a
tmComp2_t.a

A test application using both components has a makefile containing the following line:

LIBS = tmComp1 tmComp2

When this makefile is invoked in debug mode, SDE2 will automatically add a **_g** suffix to the library name and the makefile will automatically pick up the debug libraries, link them together with the test application and generate a final executable. The same is true for assert mode with an **_a** suffix, trace mode with a **_t** suffix and retail mode with no suffix.

Building in a different compilation mode can be forced using **_force_retail**, **_force_assert**, **_force_trace and _force_debug** targets, for example,

gmake all _force_debug

will always build in debug mode, overriding the value of **REL_SUFFIX**.

Mixing debug, assert and retail libraries is possible, but requires SDE2 expertise. More information regarding this feature can be found in the *Section 3.13.4.2, Mixing debug/assert/retail/trace diversities and build_exe.pl* on page 73 and *Section 3.15.4, Component diversity.mk* on page 99.

3.4.7 Warning levels

There are four warning levels within SDE2. The warning level is determined by the environment variable **_TMTGTWARNINGS**. The following four values for **_TMTGTWARNINGS** are supported:

- 0 – no warnings
- 1 – only severe warnings
- 2 – important warnings
- 3 – almost all warnings

If **_TMTGTWARNINGS** is not set, level 3 is accepted as default.

In some cases, it is not possible to suppress warning messages.

There are two types of warning messages. Microsoft compilers have their own warning levels and we mapped our levels to Microsoft's ones. GNU-like compilers support a number of warning options, you can find more information at **http://gcc.gnu.org/onlinedocs/gcc-3.0.2/gcc_3.html#SEC7**. We made the following characterization for GNU-like compilers:

- Severe warnings: **-Wparentheses**, **-Wuninitialized** (for debug mode)
- Important warnings: **-Wimplicit**, **-Wreturn-type, -Wunused**
- All warnings: **-Wall, -Wshadow, -Wpointer-arith, -Winline, -Wundef, -Wstrict-prototype**s (for C sources).

The test example for warnings is **tmComp8/tst/Tst1**.

3.5 Diversities

In general terms, a diversity is any "switch" in the source code that allows the same source code to behave differently in various situations. Some examples include endianness and CPU type. Programmers tend to be familiar with diversities handled using the **#ifdef** construct.

SDE2 provides a sophisticated means of handling this type of variation in code. SDE2's diversities go beyond the traditional **#ifdef** by providing comprehensive support for the simultaneous existence of libraries compiled with the various settings. For example, the debug and the assert versions of a library can be built serially and both will exist in an appropriate directory.

Diversities are an important part of programming techniques. The best reusable component is a component without any diversity. In many cases, diversities are indispensable.

Four types of diversities are distinguished in SDE2: *component*, *interface*, *run-time* and *BSP* (board support package) diversity. These are described in the next four sections.

3.5.1 Component diversity

Some components can be built for user-defined flavors. An example is a Teletext library with the options to compile it with a large or small cache. The code would look like:

```
/* teletext.c */
#if TXTSMALL
/* Small cache size */
#define TXT_CACHE_SIZE 10
#elif TXTLARGE
/* Large cache size */
#define TXT_CACHE_SIZE 1000
#else
#error
#endif

static txtCacheEntry txtCacheEntries[TXT_CACHE_SIZE];

...
```

When compiling this library the result can be a library with small or large cache. So, two new flavors or configurations of the library are introduced (next to the standard configurations as described in *Section 3.2.1, Selecting a configuration* on page 17).

In the makefile, there are three ways to achieve these diversities:

- Extend the makefile with a **_TMDIVERSITY** selection (see *Section 3.5.1.1, Extend the makefile with a _TMDIVERSITY selection* on page 37).
- Use a **diversity.mk** file (preferred approach, see *Section 3.5.1.2, Use diversity.mk* on page 38).
- Do a recursive make (see *Section 3.5.1.3, Recursive make* on page 39).

3.5.1.1 Extend the makefile with a _TMDIVERSITY selection

The environment variable **_TMDIVERSITY** can have a user-defined value. By default this variable is not defined. In case the compilation of a component is required with user-defined diversities, the value should be a string of the form **_<string 1>_<string 2>_ ..._<string n>_**

Here, <string i> is a field (string without spaces, underscore (_) or tilda (~)) indicating a certain diversity.

In this case, before make is invoked, the variable _TMDIVERSITY is set to _txtsmall_<xxx>_ (indicating small cache). The next step is extending the makefile of the component with the following lines:

```
#-------------------------------------------------------------------
# Find string '_txtsmall_' or '_txtlarge_' from the diversity environment
# variable _TMDIVERSITY
#-------------------------------------------------------------------
ifeq ($(findstring _txtsmall_, $(_TMDIVERSITY)),_txtsmall_)
LOCAL_CFLAGS= -DTXTSMALL
LIB_SUFFIX= _txtsmall
endif
ifeq ($(findstring _txtlarge_, $(_TMDIVERSITY)),_txtlarge_)
LOCAL_CFLAGS= -DTXTLARGE
LIB_SUFFIX= _txtlarge
endif

...
all: diversity configuration lib

diversity:
ifeq ($(LIB_SUFFIX), )
        @$(ECHO) "_TMDIVERSITY must contain _txtsmall_ or _txtlarge_ "
        @exit 1
endif
```

After that, make will build the library for the correct diversity. The result is a library with the name,

lib<NAME><LIB_SUFFIX><REL_SUFFIX>.<EXT>

Here NAME is the name of the library, REL_SUFFIX is _g, _a, or "". The EXT is a or lib. Also LIB_SUFFIX is a variable used by SDE2 that can be set by the component makefile. For example, libtmtxt_txtsmall_g.lib

An example of the use of the _TMDIVERSITY flag can be found in the example component tmComp2 of the SDE2 distribution that has user-defined diversity.

3.5.1.2 Use diversity.mk

The content of diversity.mk is described in Section 3.15.4, Component diversity.mk on page 99. It is a natural extension of the first case to use _TMDIVERSITY. The main idea is to remove the diversity definitions from the makefile in order to make them accessible directly to other components via SDE2. The process is illustrated by the following example.

Let tmComp10 require tmComp8 built either in multiprocessor or in single processor mode depending on the current value of _TMDIVERSITY. So, an ifeq construction can be applied in tmComp10 using the approach from the previous section for tmComp8. However, if several components require tmComp8, every component has to have this section. It is more efficient to have this information only once, i.e., in the file tmComp8/diversity.mk (see the

example in Section 3.15.4). Now, **tmComp10** has in its **LIBS** section, **LIBS=tmComp8**. Further, SDE2 includes **tmComp8/diversity.mk** if it is present. **tmComp8/diversity.mk** sets the values of **_tmComp8_SUFFIX** and **_tmComp8_DIVERSITY**. These values are filtered out by SDE2 yielding the final diversity. This mechanism may be complex, but a user only has to:

- Write the **diversity.mk** file
- Set the **LIBS** section in all component that require this component

As an example see **tmComp8/diversity.mk** at Section 3.15.4.

3.5.1.3 Recursive make

Another way to deal with component diversity, is to modify the makefile in such a way that the makefile automatically creates both flavors of libraries. This is the approach which must be used if you always need to produce more than one library per component with different flavors. In fact **make** must be called at least twice by the component's makefile. This is called a recursive make.

When doing a recursive make, it is important that the **C_SOURCES** line at the top-level invocation of the make is empty (since GNU **make** otherwise starts to generate dependencies). The standard variable **C_SOURCES** is renamed to **C_SOURCES_gen**. In the recursion the value of **C_SOURCES_gen** is assigned to **C_SOURCES**. To generate the name of the image, a **LIB_SUFFIX** variable needs to be specified as described in the previous section. In the **all** target the recursion is created. The first **make** invocation generates the small cache version of the library, the second invocation, the large cache processor version. Below is the **tmComp7** makefile (without comments):

```
DIR_LOCAL   = comps/tmComp7
include $(_TMROOT)/sde/environment.mk

CXX_SOURCES  =
C_SOURCES_gen  =\
        src/tmComp7.c \
        src/$(_TMTGTCPUCLASS)_$(_TMTGTOSCLASS)/tmComp7PlatformSpecific.c

REQUIRES  =
DIR_INCLUDE  = src
LOCAL_CPPFLAGS  =

LIB_SUFFIX=$(subst lib,,$(COMP7_LIB))
C_SOURCES_sp = $(C_SOURCES_gen) # add here sp specific src/tmComp7Sp.c
LOCAL_CFLAGS_sp = $(LOCAL_CFLAGS) -DTMFL_SCOPE=TMFL_SCOPE_SP
C_SOURCES_mp = $(C_SOURCES_gen) # add here mp specific src/tmComp7Mp.c
LOCAL_CFLAGS_mp = $(LOCAL_CFLAGS) -DTMFL_SCOPE=TMFL_SCOPE_MP

all: configuration FORCE
        @$(ECHO) making multi processor version
        @$(MAKE) "COMP7_LIB=lib_mp" "C_SOURCES=$(C_SOURCES_mp)" \
                "LOCAL_CFLAGS=$(LOCAL_CFLAGS_mp)" lib
        @$(ECHO) making single processor version
        @$(MAKE) "COMP7_LIB=lib" "C_SOURCES=$(C_SOURCES_sp)" \
                "LOCAL_CFLAGS=$(LOCAL_CFLAGS_sp)" lib
```

```
ifneq ($(DIR_CONFIG),_)
include $(DIR_SDE)/$(DIR_CONFIG)/makelib.mk
endif

ifeq ($(strip $(TMFL_CPU_IS_TM)),1)
$(DIR_INTERM)/src/tmComp7.$(_SDE_O) : TARGET_CFLAGS+=-O4
endif
```

3.5.2 Complex interface diversity

Some development projects use an interface-based way of working. Interfaces are put in the `intfs` directory. Components are introduced that implement those interfaces.

In some cases, the interfaces are abstract and become concrete when connected to a certain component during compilation. Clients using those abstract interfaces must identify in their makefile what component is making the abstract interface concrete. This is done by using the **PROVIDED_BY** keyword in the **REQUIRES** section of the makefile, for example,

```
#-------------------------------------------------------------------
# Required components
#-------------------------------------------------------------------
REQUIRED_INTERFACES = \
        tmReal PROVIDED_BY tmRealFloat

PROVIDED_INTERFACES =

REQUIRES = $(REQUIRED_INTERFACES)
```

An example of an abstract interface that is implemented by a component can be found in `comps/tmRealFloat` in the SDE2 distribution. An example of a client using the abstract interface can be found in `comps/tmComp3`.

3.5.3 Run-time diversity

Some components offer a so-called run-time diversity. This type of diversity uses run-time information during the build/run process. There are different definitions of run time and we use a broad one. We assume that the run-time diversity is something which happens after SDE2 component delivery. There is another definition which states that run-time diversity is something which happens in run time, for example reading user input, reading color of the window, etc. This definition is not used within SDE2 because there is no need to define such diversity within SDE2.

A good example for run-time diversity within SDE2 is when the source code is split into two parts: a part that may be modified and recompiled by clients (the run-time diversity part), and a part that must not be modified by the component.

The normal (not-configurable) code is put in the `src` directory of the component. For the configurable code (run-time diversity part) a new directory `cfg` next to the `src` directory is introduced. In the makefile of the component a new line is introduced:

CFG_SOURCES = <sources that may be re-compiled by clients>

For example,

CFG_SOURCES = cfg/tmComp6_Cfg.c

Note: Only a **c** extension for configurable sources is allowed.

When the makefile is executed, all code is compiled and linked to a library. The library also contains the object code of the **CFG_SOURCES**. After that the SDE2 copies the configurable source files defined in **CFG_SOURCES** to

<_TMTGTBUILDROOT>/comps/<component name>/cfg

Applications (executables) that can use the default configuration can use the component in the normal way. Applications that want to configure the component in another way, have to recompile the configurable part of the component. After that, SDE2 will take care that the executable contains the reconfigured object code instead of the original object code. Note that this mechanism will not work for DLLs.

Therefore, the makefile of an application/executable typically contains the following lines of code (also see **tmComp6/tst/Tst1** in the SDE2 distribution):

```
# tmComp6Cfg.c is the configurable code of component 6.
C_SOURCES      = \
            src/test.c \
            ...
            tmComp6Cfg.c

#-------------------------------------------------------------------
# Extend the VPATH to find the configuration files
#-------------------------------------------------------------------
DIR_CFG_SOURCES=$(_SDE_DIR_REL_TO_LOCAL_ROOT)/comps/tmComp6/cfg
VPATH += $(DIR_CFG_SOURCES)
...
DIR_INCLUDE                  = src $(DIR_CFG_SOURCES)
```

The original component **tmComp6** is compiled with header file **inc/sys/tmSysCgf.h**. For recompilation of the configurable code, i.e., **tmComp6_Cfg.c**, the following include directory is introduced: **$(DIR_CFG_SOURCES)** which is the directory containing **tmComp6_Cfg.h**. **tmComp6/tst/Tst1/src** contains a modified header file, **tmSysCfg.h**, that is included by **tmComp6_Cfg.h**.

3.5.4 BSP diversities

Linking libraries and objects into an executable is the most diverse part of SDE2. This final link process is very dependent on the environment. Making a TriMedia executable is significantly different than making a **mips-pSOS** image for a standard Algorithmics board.

The link rules for a certain platform are found in the **maketarget<_TMBSL>.mk** file of the directory **sde/<_TMTGTCPUCLASS>_<_TMTGTOSCLASS>**. For most configuration classes a default **maketarget.mk** is supplied (_TMBSL is empty). For the **mips-psos** configuration class the file **maketarget_p4032.mk** (**_TMBSL=_p4032**) is supplied. This file includes a **maketarget_generic.mk** that defines the linking rules for a P4032 Algorithmics board.

Having a board other than the one supported by the SDE2, means making your own **maketarget<_TMBSL>.mk** file. The content of such a file is discussed in *Appendix G*, *The maketarget<_TMBSL>.mk files*. If the new link rules (more or less) match the original SDE2 link rules, it may be wise to create a new **maketarget<_TMBSL>.mk** makefile that does an include of a generic makefile that contains the common parts of the link process, see the **maketarget_p4032.mk** file of the **sde/mips_psos** directory.

When building is required for a new board, the environment variable **_TMBSL** is set to the new value. During the build, the new **maketarget<_TMBSL>.mk** will be selected for building a final image.

A good example for BSP diversity, is the **maketarget_dvp1.mk** makefiles (one per platform) that are delivered with SDE2.

The following table lists the standard SDE2 makefiles (**maketarget<xx>.mk**) for making an image or executable per configuration class.

Table 3-9: Standard makefile per configuration class

Configuration class	maketarget<xx>.mk	Description
mips_psos (ISI Diab Data compiler)	**maketarget_p4032.mk** **maketarget_dvp1.mk**	Makefile for making a P4032 Algorithmics board image
All other platforms	**maketarget.mk** **maketarget_dvp1.mk**	Makefile for making a default and DVP1 platform-specific images

3.6 Dynamic link libraries (DLLs)

SDE2 supports DLLs for **arm_ce**, **armgnu_linux**, **mips_ce**, **mipsgnu_linux**, **tm_psos**, **x86_nt**, **x86_ce**, **.**, **x86gnu_nullos** and **x86gnu_linux**.

3.6.1 Generation

The generation of a DLL is started if the **EXPORTS** makefile variable is not empty. In this case SDE2 generates static libraries and DLLs. The example component **Comp8** illustrates the way DLLs are generated by SDE2.

If the export function (variable) is defined by

```
void tmComp8Print(int i);
```

then in the makefile you have:

```
EXPORTS = tmComp8_Print
```

The DLL names are as follows:

- **tm** – The name is the same as a library name and the extension is **dll**.
- **nt**, **ce** – The name is the same as a library name without a **lib** prefix and the extension is **dll**. Additionally the.**lib** and.**exp** files are generated with the DLL's name.
- **linux**, **x86gnu_nullos**– The name is the same as a library name with a **lib** prefix and the extension is **so**.

Note: You must add **extern C** in your sources if you compile in C++ mode.

3.6.2 DLL generation options

Different DLL diversities can be set using the following environment variables:

- **_SDE_DLL_OPTIONS** – If you put your options here, they will be at the beginning of all options excluding **_SDE_LBOPTS**.
- **LOCAL_DLLFLAGS** – If you put your options here, they will be at the end of all options.

So, the order of DLL options is:

<_SDE_LBOPTS><_SDE_DLL_OPTIONS><All export symbols, platform-specific stuff, libraries and DLLs> <LOCAL_DLLFLAGS>.

Note: If you execute the binary executable from the SDE2 location, you have to set the DLL path to your **PATH** environment variable.

3.6.3 DLL directory structure

This DLL directory structure has been introduced to give good support for diversity mechanisms. The relation between different DLLs is established at compile time, so it is not possible to link different DLLs at link time if they do not contain the proper link between them made at compile time. Therefore, sets of DLLs for different diversities have to be supported. This is done by having one (or a set of) fixed DLL name per component and placing this DLL in a subdirectory of the generated **lib** directory. The name of the directory is determined by the concatenation of the values of **_TMDIVERSITY** (sorted by items separated with underscores; without trailing underscores, if present) and **REL_COMPS_SUFFIX** (see *Table 3-12 Variables for any makefile*). If both are empty, the current platform-specific **lib** directory is selected. For example, for

_TMDIVERSITY=_flo_sp_

REL_COMPS_SUFFIX=_g

the directory name is **_flo_sp_g**

For,

_TMDIVERSITY=_kernel_

REL_COMPS_SUFFIX=

the directory name is **_kernel**

For an **x86_nt** compilation, assume there are two components – **tmOsal** and **tmcr** – and two diversities to compile with, **_mp_** and **_sp_** (both in debug mode). This will yield the following output directory structure:

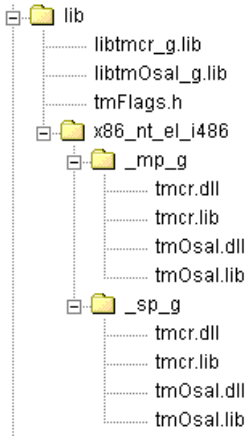


Figure 3-6: Example x86_nt compilation with two components and two diversities

This directory structure also applies in the case of a binary release.

3.6.4 Usage

You can use DLLs with the following statements in the makefile's DLL line:

```
REQUIRES = tmComp8 tmComp10
  "
LIBS = tmComp8 \
    tmComp10
```

Optionally you could have

```
_tmComp8_LOADED=immediate
_tmComp10_LOADED=deferred
_tmComp10_SUFFIX=_r
```

The **REQUIRES** statement contains a list of required LIBs and DLLs within SDE2.

If required, the DLLs are searched for in two places:

- In the (sub)directory of the generated lib's directory (see above)
- In the same (sub)directory of the binary release lib directory.

Note: The application or component links with the DLL only in the case that this DLL is present (in the release tree or in the component binary release). Otherwise it links with the static library.

There are a few different DLL load modes for **tm_psos**. Quite often more than one is used. To facilitate this **tm_psos** requirement, the keyword **_<CompName>_LOADED** is introduced in SDE2. If it is present, the required DLL in the specified mode is loaded. If it does not exist, the default load mode **immediate** for **tm_psos** compilation is taken.

The _<CompName>_LOADED keyword is not used for x86_nt, x86_ce, arm_ce and mips_ce.

If you use external DLLs you can set their locations and names in EXTERNAL_DLLS. For example,

```
EXTERNALS_DLLS = c:/my_1_location/my_1_dll e:/my_2_location/my_2_dll
```

The usage of external DLLs will make your software less reusable.

Writing DLL source files does not always require having a DLL entry point DllMain. However, in the current SDE, this entry point is always required for the x86_ce platform, see tmComp8. More information about the DLLs is available at http://www.cygwin.com/docs.html.

3.6.5 C++ support and DLLs

SDE2 supports C++. However, we generate DLLs only if EXPORTS is not empty. This might be the case in C++, for example, if we would like to export only classes. In this case the export definition is in the header file. In order to generate DLLs, we need to make EXPORTS in the component makefile non empty. This can be accomplished as follows:

```
empty =
EXPORTS = $(empty) $(empty)
```

See tmComp9 as an example for C++ support.

3.6.6 Suppressing DLLs

DLL generation is suppressed if you invoke the _sde_suppress_dlls target, for example,

```
gmake all _sde_suppress_dlls
```

3.7 Libraries and the LIBS section

To illustrate the relevance of this concept, consider the following example.

You have a large number of components and you change something in one of these components with the consequence that it requires another BSP library, it has another diversity, etc. It is easy to assume that most of the developers writing applications based on this component are not aware of the change. Now you have two options:

- Inform all the users of the change and instruct them to change their executable-makefile, editing its LIBS section.
- Have SDE2 handle this case.

The first option may be problematic, therefore we prefer to have SDE2 handle this case. Here it is described how it is used and how the implementation is made.

3.7.1 How is the LIBS section used?

The only thing you need to know about **LIBS** sections is that if you use functions from another component, then this component should be placed in the **LIBS** section. Typically, the **REQUIRES** section consists of required component and required interfaces. The **LIBS** section consists of (subset of) required components, for example,

```
REQUIRED_COMPONENTS = tmA tmB tmC
REQUIRED_INTERFACES = tmD tmE
REQUIRES = $(REQUIRED_COMPONENTS) $(REQUIRED_INTERFACES)
LIBS = $(REQUIRED_COMPONENTS)
```

There cannot be circular-dependent **LIBS** sections.

Now we are going more deeply into the compilation process and components' dependency generation.

3.7.2 DependsOn relation

We say that the component/executable A *DependsOn* component B if B is needed in order to:

- Compile A's DLL
- Link A or an executable which *DependsOn* A. During the linking process, library B is placed after library A.

So, we introduce the *transitive* relation DependsOn and this relation has its physical implementation in the **LIBS** section of the component makefiles. In all component makefiles it is necessary to mark the dependencies of the component. For example, for the **comps/tmComp10/makefile**:

```
LIBS=tmComp8 tmComp3 tmComp2
```

The static library compilation process does not depend on this line. The goal (permanent from SDE2 1.2) that depends on this is **_sde_libs**. The purpose of **_sde_libs** is to print the required libraries to a file **<CompName>.l**. This file is located in the directory **$(_SDE_DIR_BUILD)/tmp/$(_SDE_LIB_CONFIGURATION)/$(_SDE_DIVERSITY)**.

The ***.l** files contain the list of all the required component (libraries) without prefixes and suffixes per *specific configuration* (processor type and class, os class, diversity, rel.mode). The component names are space separated, for example for **_mp_g/tmComp10.l**,

```
tmComp8 tmComp3 tmComp2
```

This is the copy of the **LIBS** statement for the specific configuration.

Per *specific configuration* means that the **LIBS** section might be configuration-dependent and for different configurations we place ***.l** files in a different directories.

The names of the libraries are determined at the linking time of the executable/DLL. These names are derived from **_SDE_IMPORT_DLLS** for DLLs and in **_SDE_LIBS_DIVERSITY** for libraries, for example (we assume that **tmComp8.dll** is present),

```
gmake var=_SDE_IMPORT_DLLS var2=_SDE_LIBS_DIVERSITY _sde_print_var_5
_SDE_IMPORT_DLLS = tmComp8
_SDE_LIBS_DIVERSITY = tmComp2_flo_g tmComp3_g
```

Having all *.l files for a given configuration, the transitive closure of the libraries can be computed. That means that a correctly ordered list of all required libraries can be created and submitted to the link line for any executable or DLL that requires one or more of them. So, the result from the DependsOn relation finally ends in the correct compilation line at link time.

Note: Do not manually change *.l files. Even if you have good knowledge of how SDE2 works, it is not a good idea to distribute your sources with instructions how to change *.l files between the compilation processes. This may only be done for debug purposes.

SDE2 is responsible for doing this. It sets all the required libraries and DLLs in the correct order in _SDE_IMPORT_LIBS and _SDE_IMPORT_DLLS. Further, _SDE_LIBS_DIVERSITY is computed on a base of _SDE_IMPORT_LIBS with the following flavors/changes:

- If _<CompName>_DIVERSITY is defined for some CompName from _SDE_IMPORT_LIBS, this is added after the component name;
- If _<CompName>_SUFFIX is defined for some CompName, then this component is used always in a fixed compilation mode;
- If _<CompName>_REPLACE is defined for some CompName, then the library of this component is replaced during the link time with the library of _<CompName>_REPLACE .

DLLs have no suffixes, they are placed in different directories for a different values of _TMDIVERSITY and _TMTGTREL .

The variables _SDE_LIBS_DIVERSITY and _SDE_IMPORT_DLLS are used for linking of:

- Executables
- DLLs for nt or ce

Thus, the LIBS section introduces link-time dependency. The REQUIRES section introduces only a compile time dependency.

3.7.3 Overriding default diversities and *.l files

There are three levels of mixing debug/assert/retail/trace modes and diversities:

1. All required libraries are built using the current value of _TMTGTREL and _TMDIVERSITY, no mixing.
2. The component can be specified that it is always required in a fixed compilation/diversity mode. As a rule, the diversity mode is derived from _TMDIVERSITY. This is specified in the component's diversity.mk file. You can do this in the following way:
   a. The value of _<CompName>_SUFFIX can be set to _g, _a, _t or _r. This means that this component will be required in a fixed (debug, assert, trace and retail correspondingly) compilation mode unless the makefile overrides this behavior. Additionally, it can be stated in the makefile that the component must be built only in this mode, by changing the value of REL_SUFFIX to _g, _a, _t or to the empty string (denoting retail mode).
   b. The value of _<CompName>_DIVERSITY is derived from _TMDIVERSITY in the component's diversity.mk file. This file is included in the final linking process, so the executable knows which diversity it needs. See Section 3.15.4, *Component diversity.mk*.

3. It can be defined that an executable or DLL requires a component with a fixed compilation mode or diversity. In this way the component's default compilation mode can be overwritten. This should be done in the makefile of the executable, for example, **_tmComp1_SUFFIX=_a _tmComp2_DIVERSITY=_mp**

In the second and third level it must be taken into account that DLLs and **.l** files are generated into the directory determined by the value of **_TMTGTREL**. So, if you have **_TMTGTREL=debug** and if you force **tmComp8** to be generated in assert mode, it will be placed in a subdirectory (e.g., **_mp_g**) which is correct because all DLLs should be in one release directory determined by the values of **_TMDIVERSITY** and **_TMTGTREL**.

If the component A, which creates a DLL, requires a component B in a different compilation mode and this is specified in A's makefile, then all the executables/DLLs which DependOn component A must specify in their makefiles this different compilation mode for B as well. This is very inconvenient. Therefore, it is better to specify this in the A's **diversity.mk**. Then all executables/DLLs will be able to read this information in SDE2 run-time.

Example present in SDE2:

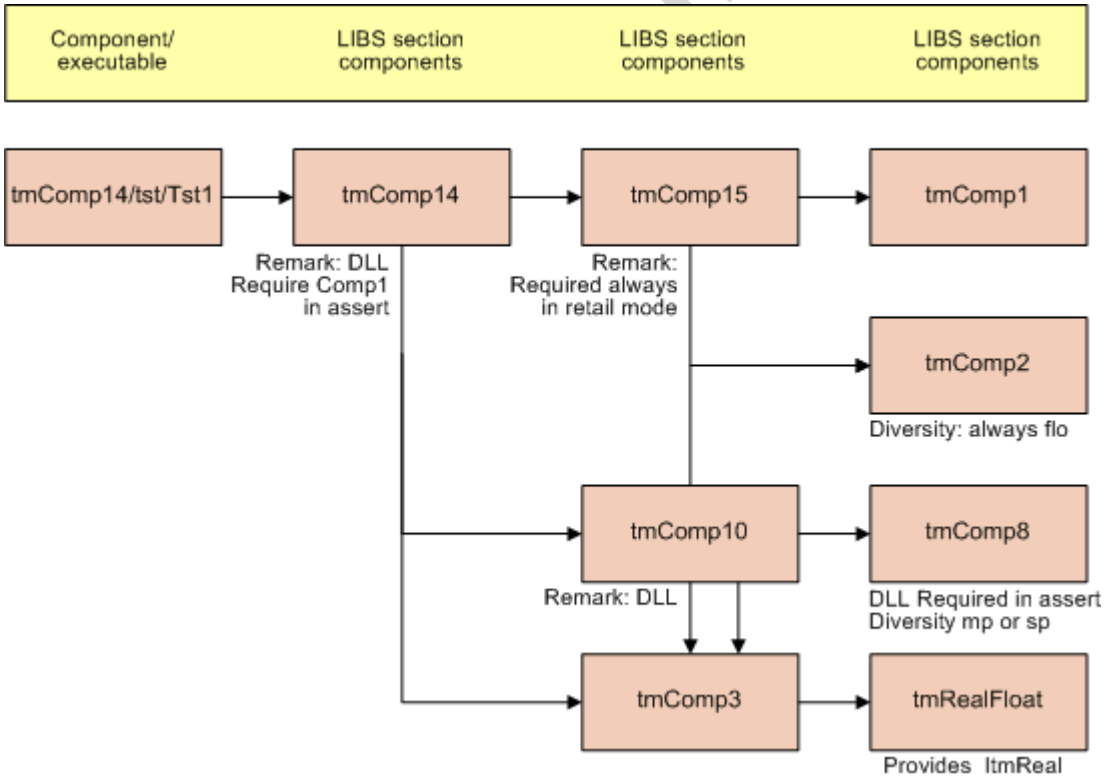


Figure 3-7: Example of DependOn relationship

Note: Makefile of **tmComp14** requires **tmComp1** to be built and used in assert mode

3.7.4 Missing *.l file(s)

If one or more *.l files are missing during the build of an executable or DLL, SDE2 will give you a warning message, for example,

```
Warning: tmRealFloat.l file(s) in c:/b_result_1206/comps/<CompName>/tmp/x86_nt_el_i486/_g is/are missing. Read
Chapter 3.7.4 of the User Manual!!!
```

and will refer to this section. If a *.l file is missing, it can cause a failure of the build process, because SDE2 cannot make the correct compilation line. This could be caused by:

- Changing one or more environment settings between the different builds;
- Directly using a binary (library) release;
- Removing the existing *.l files;

This can be corrected easily by one of the following two approaches:

- If one of the component's *.l file is missing, you can go to this component's directory and type **gmake** (or **gmake _sde_libs**);
- If more of the component's *.l files are missing, it is better to use a **build_exe.pl** script in a following way:

```
perl build_exe.pl -l <your component/executable>
```

This script will find recursively all required components and it will build all the *.l files for you. For more information about the scripts, see *Section 3.1, Common directory structure* on page 10.

3.7.5 Promotion of the interface

Let's say that you have a component A. Due to some reasons, you have to promote the interface of this component to the **intfs** directory. There are few steps you have to complete.

1. Create in the **intfs** directory subdirectory **ItmA**.
2. Copy the public header files from **tmA/inc** to **ItmA/inc**.
3. Delete the files in the **tmA/inc** directory.

The biggest advantage of these approaches is that your components/executables, which require **A** will still work. They will require the interface in **ItmA** and they will use the library of component **A** as a default without any change of the makefile.

Of course, you can define your interface with a unique name and start using it without a default implementation.

3.7.6 Replacing the libraries and DLLs

During the development process, for some reason you may use the interface of one component. But during the linking, you need another implementation of the same interface. SDE2 offers the possibility to replace one component with another during the final linking process. Both components must provide (and require) the same interface. You can do this in your executable makefile as follow:

_tmComp17_REPLACE = tmComp16

If you do this, you are responsible for the following:

- Both components should have (provide/require) the same interface.
- In the case that one generates a DLL, the other should do this too. Moreover, both should have the same **EXPORTS** section;
- If one has a binary release (for some diversities), the other must have a binary release as well (for the same diversities).

During the replacement, the flavors (suffixes) of the component are not changed.

3.8 SDE_in_SDE

The concept of **SDE_in_SDE** describes a system of structuring a component hierarchically. In this section the concept will be described with an example (see *Section 3.8.1, Example component structure for SDE_in_SDE* on page 50).

The MoReUse standard allows you to organize the internal structure of a component's **src** directory according to your own preferences. SDE2 also does not prescribe how the **src** directory should be organized. However, SDE2 supports an extra feature called **SDE_in_SDE**.

The basic principle is that the directory **src** contains a subdirectory **comps** that contains a number of subcomponents.

The delivery of SDE2 contains an example component for **SDE_in_SDE (tmComp5)**.

3.8.1 Example component structure for SDE_in_SDE

Let's look at the following example:

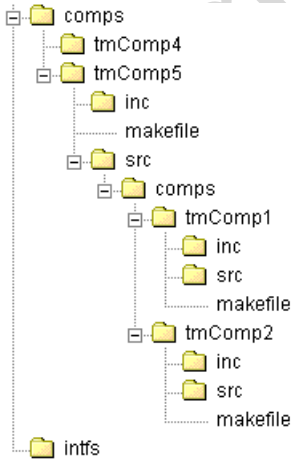


Figure 3-8: Example component structure for SDE_in_SDE

Here, tmComp1 and tmComp2 can be compiled separately by using gmake in their respective directories. We need not have to select uniqe names for the sub components, as in the case of the main components. However, it is recommended to use the unique names. We can also build the sub-components automatically, by building the main components. In order to integrate the sub-components in the whole build process, some rules have to be adhered to. These are listed and explained in the next section.

3.8.2 Rules for SDE_in_SDE

- The sub components tmComp1 and tmComp2 would have their own makefiles. Hence they can by compiled separately, as done in the case of main components.
- Any external component (eg: tmComp4) does not have access to the (libraries and object files of the) internal sub components tmComp1 and tmComp2.
- The sub components tmComp1 and tmComp2 can have access to any external component (eg: tmComp4）. This has to be set in their REQUIRES sections of the sub component's makefile. Further, the LIBS section of the main component tmComp5 must be set to tmComp4. Explanation: Source file in the `src` directory of tmComp5 requires tmComp4, which must be marked in the.l files.
- The main component tmComp5 has access to the libraries and objects of the sub components tmComp1 and tmComp2. However, this should be marked explicitly in the main components makefile. This could be achieved by adding the following line to the main component's make file ;

```
DIR_INCLUDE = src/comps/tmComp1/inc src/comps/tmComp2/inc
```

- The generated libraries of the sub components tmComp1 and tmComp2 are located in `$(_TMTGTBUILDROOT)/comps/tmComp5/lib/$(_SDE_LIB_CONFIGURATION)`. They have to be in a location different from the release directory.
- The main component tmComp5 is responsible for building the sub components tmComp1 and tmComp2. Typing gmake will build the sub components tmComp1 and tmComp2 first and after that the main component tmComp5.
- If the user wants, he can get a copy of the sub component's library by setting _TMTGTCOPYLIB to **1**. The library of the sub components are copied to ../$(DIR_LOCAL)/src/comps/<sub component name>/lib/`$(_SDE_LIB_CONFIGURATION).However, this copied library of the sub components can not be automatically used in the build process.`
- Normally, the binaries of the sub components have to be embedded in the binary of the main component by including the sub component's object files while linking the library of the main component.. This is shown in the following example.\The following

```
_SDE_OBJECTS += \
$(_SDE_DIR_BUILD)/src/comps/tmComp1/tmp/ \
$(_SDE_LIB_CONFIGURATION)$(_SDE_ARSUFFIX)/src/ \
tmComp1.$(_SDE_O)\
```

example shows the

3.8.3 Defining SDE_in_SDE in your makefile

It is easy to define your sub components wiith in the main component in SDE2.

The following steps have to be followed while building a component having the sub components:

- Build the sub components first.
- Include their object files to **_SDE_OBJECTS variable of the main component.**
- Set **DIR_INCLUDE** to required header files of the sub components in the main component's makefile
- Build the main component
- The following example shows the content of the main component's makefile

```
DIR_LOCAL= comps/tmComp5
#*****************************************************************************
# Do not change the following include
#*****************************************************************************
include $(_TMROOT)/sde/environment.mk
#-----------------------------------------------------------------
# Source environment variables
#-----------------------------------------------------------------
CXX_SOURCES =
C_SOURCES=\
        src/tmComp5.c
#-----------------------------------------------------------------
# Required components
#-----------------------------------------------------------------
REQUIRES    =
LIBS    = tmComp4
#*****************************************************************************
# Do not change this
#*****************************************************************************
all: configuration FORCE
        @$(ECHO) making internal component tmComp1
        @$(MAKE) -C src/comps/tmComp1
        @$(ECHO) making internal component tmComp2
        @$(MAKE) -C src/comps/tmComp2
        @$(ECHO) making main component tmComp5
        @$(MAKE) lib

main: configuration lib

Directory where the sde_in_sde includes are stored
#-----------------------------------------------------------------
DIR_INCLUDE= src/comps/tmComp1/inc \
                src/comps/tmComp2/inc


ifeq ($(findstring _flo_,$(_TMDIVERSITY)),_flo_)
_tmComp2DIV := _flo
else
ifeq ($(findstring _int_,$(_TMDIVERSITY)),_int_)
_tmComp2DIV := _int
endif
endif


_SDE_OBJECTS += \

$(_SDE_DIR_BUILD)/src/comps/tmComp1/tmp/$(_SDE_LIB_CONFIGURATION)$(_SDE_ARSUFFIX)/src/tmComp5C
omp1.$(_SDE_O) \

$(_SDE_DIR_BUILD)/src/comps/tmComp2/tmp/$(_SDE_LIB_CONFIGURATION)$(_tmComp2DIV)$(_SDE_ARSUFFI
X)/src/tmComp5Comp2.$(_SDE_O)

#*****************************************************************************
# Do not change the following include
#*****************************************************************************
ifneq ($(DIR_CONFIG),_)
include $(DIR_SDE)/$(DIR_CONFIG)/makelib.mk
endif
```

- For sub components, the following sections are needed before including **environment.mk**:

```
DIR_LOCAL      = src/comps/tmComp1
SDE_IN_SDE     = comps/tmComp5
```

Here **DIR_LOCAL** is set to the name of the component with an **src/comps/** prefix.

SDE_IN_SDE contains the name of main component.

- The following example shows the content of the sub component's makefile

```
DIR_LOCAL= src/comps/tmComp1
SDE_IN_SDE= comps/tmComp5
#*****************************************************************************
# Do not change the following include
#*****************************************************************************
include $(_TMROOT)/sde/environment.mk
#-----------------------------------------------------------------
# Source environment variables
#-----------------------------------------------------------------
CXX_SOURCES =
C_SOURCES=\
        src/tmComp5Comp1.c
#-----------------------------------------------------------------
# Required components
#-----------------------------------------------------------------
REQUIRES       = tmComp4
LIBS =
#-----------------------------------------------------------------
# Directory where the 3rdparty includes are stored
#-----------------------------------------------------------------
DIR_INCLUDE=

#*****************************************************************************
# Do not change this
#*****************************************************************************
all: configuration lib

#*****************************************************************************
# Do not change the following include
#*****************************************************************************
ifneq ($(DIR_CONFIG),_)
include $(DIR_SDE)/$(DIR_CONFIG)/makelib.mk
endif
```

3.8.4 SDE_in_SDE and gmake clean

If you use **gmake clean** in a subcomponent, it cleans only the files generated by the subcomponent. If you execute **gmake clean** in a main component, it cleans the files generated by the main component, but not the files from the subcomponents. So, if you want to clean all the files from your main directory, you have to define **clean** rule in yourmain component's makefile.:

```
clean::
        @$(MAKE) -C src/comps/tmComp1 clean
        @$(MAKE) -C src/comps/tmComp2 clean
```

3.9 Multiproject SDE2

The multiproject feature in SDE2 can be summarized as working with multiple **comps** directories. It may happen that a development team develops a component that depends on components (source or binary release) of other projects and to which the development team has (read) access. This usually will happen using a configuration management tool, but that is not relevant for the principle.

A possible solution for the multiproject problem is to copy all component data of the different development teams that is relevant for you into your own **comps** directory. This solution however has a number of disadvantages:

- Your project data usually are under control of a configuration management tool and you do not want to mix it with other project's data.
- If several projects use the same data, there will be several identical copies in the network.
- Copying (of the external project) may be needed quite often to keep consistent with the other project.
- It is an inelegant way involving ad hoc actions.

SDE2 offers a solution to handle this situation, which is explained in the following section.

3.9.1 Multiproject implementation in SDE2

Let us look into the following example of multiple project structure.

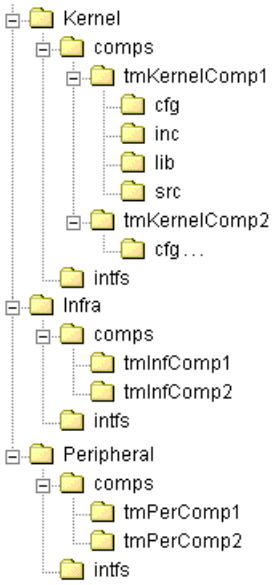


Figure 3-9: Example of a multiple project structure

This example also considers one global SDE directory containing the SDE2 release, the `inc` directory and the `project` directory

Figure 3-10: Global SDE directory

Lastly, this example considers that the implementation is to be hidden in SDE2. There are two files generated from **prjlist.txt**. The files **loc_list.txt** and **loc_list.mk** are located in `<_SDE_TMTGTBUILDROOT>/project`.

As a rule, if more than one component with the same name is present in different projects, SDE2 is working with the last specified component (i.e., in the last specified project in **prjlist.txt**).

The file **prjlist.txt** contains space-separated locations of the projects, for example,

c:/KERNEL e:/INFRA f:/ready/PERIPHERIAL

The file **prjlist.txt** may also contain environment variables. If you change the values of these variables (e.g. to update **loc_list.*** files), you are responsible for **touch**'ing the file. For example for the file **prjlist.txt,**

$_TMROOT $_TMREQPRJ

The file **loc_list.txt** contains names of the components (interfaces) and their locations, space separated, for example,

tmKernelComp1 c:/KERNEL/comps/tmKernelComp1 tmKernelComp2 c:/KERNEL/comps/tmKernelComp2 tmInfComp1 e:/INFRA/comps/tmInfComp1 ...

The file **loc_list.mk** contains the assignment of the makefile variables for the components' (interfaces) locations, for example,

```
_tmKernelComp1_DIR := c:/KERNEL/comps/tmKernelComp1
_tmKernelComp2_DIR := c:/KERNEL/comps/tmKernelComp2
_tmInfComp1_DIR    := e:/INFRA/comps/tmInfComp1
...
```

The **loc_list.*** files are regenerated when components are added. If you receive the following message,

SDE2 cannot find the definition of **_<your comp>_DIR**. Update your **prjlist.txt** or **loc_list.*** files.

then you have to regenerate your **loc_list.*** files. The generation is best done in the following way:

- Delete your **loc_list.*** files (via **clean_all** or manually)
- Execute **gmake makelist**

The target **makelist** is responsible for regeneration of the **loc_list.*** files. In the following cases **loc_list.*** files are updated:

- If they are not present
- If one of the directories or **prjlist.txt** has a later date, i.e., we have changed something in the directory structure
- **_<CompName>_DIR** is invalid. This happens if we switch between Unix and NT.

All these cases cover majority of all possible cases when we have to regenerate **loc_list.*** files.

In order to ensure the backward compatibility with the previous versions of SDE2, the following behavior is defined. If the file **prjlist.txt** is not present, it is created and contains the value of **_TMROOT**. In this way, the directories **$(_TMROOT)/comps** and **$(_TMROOT)/intfs** are considered by default.

The advantages of this solution to the multiproject problem that is implemented in the SDE2 approach are:

- No superfluous copies of component data.
- More structured, project-oriented development.
- Development of different projects simultaneously.
- Easy (source) release per project
- It is always possible to move/restructure your components in a different structure without harming their functionality. For example, you can move all of them to one **comps** directory.

- The development process is independent of the project structure. You write

  REQUIRES = tmKernelComp1 tmInfComp1

  LIBS = tmKernelComp1

  and you do not care where **tmKernelComp1** and **tmInfComp1** are located.

A disadvantage with SDE2's solution of the multiproject problem is that it is necessary to use absolute include paths during the compilation of the component, which can be quite long. But, because nearly all compilers/shells[5] have no problems with long paths, the effect of the disadvantage is marginal.

When you develop your components, take into account that they know nothing about the locations of other components and interfaces, so do not include header files with absolute locations or even with relative locations using a back path (i.e.**../../comps/tmComp1/inc**).

3.9.2 Practical recommendations for using multiproject SDE2

The above section, *Multiproject implementation in SDE2*, explains how to work with multiple projects in SDE2. If you have more than one project and each project needs to be built in different configurations, it is recommended you follow the methods given below.

- Edit the **prjlist.txt** file present in **$(_TMROOT)/project** and add an environment variable **$_TMREQPRJ**. For example, **$(_TMREQPRJ) $(_TMROOT)**
- Create a separate batch file for each project configuration, to initialize the corresponding environment variables for the required configurations.
- Add the definitions of the variables **$(_TMREQPRJ)** accordingly in each project configuration-specific batch files.
- Also, define the variable **$(_TMTGTBUILDROOT)** differently in each batch file.
- A component should be stored in only one particular location. Do not keep multiple copies of the same component.
- Now you can use SDE2 simultaneously for different projects based on different configurations, by running the appropriate batch files.

This approach ensures that

- A single SDE2 installation is enough to perform multiproject operations and the user need not have duplicate copies of any component.
- There is no interference between any two different projects, in that different **loc_list.*** files are generated and the components are built in different locations because **$(_TMTGTBUILDROOT)** is set differently for different projects.

3.10 Binary release

If you deliver your component together with your libraries, DLLs and/or object files, then you make use of a *binary release*. SDE2 supports this process. SDE2 does not support a binary release for ***.l** files, they can be generated easily by the **build_exe.pl** script.

5. Some UNIX shells support up to 5000 characters, others support 16,000.

3.10.1 Generation

SDE2 supports two types of binary releases:

- Binary release of libraries and DLLs
- Binary release of object files

3.10.1.1 Binary release for libraries and DLLs

You can generate a binary release for libraries and DLLs, if you set _TMTGTCOPYLIB to 1. As a result, SDE2 will copy all the generated libraries and DLLs into your components' `lib` directory. For example for a WinNT host and target,

```
..\comps\tmComp8> set _TMTGTCOPYLIB=1
..\comps\tmComp8> gmake
OR
..\comps\tmComp8> gmake _sde_bin_rel
```

This produces the following structure:

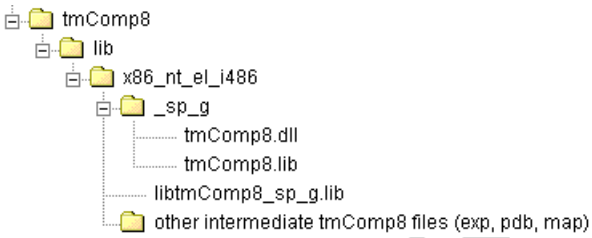


Figure 3-11: Global SDE directory

SDE2 prints the message:

```
copying the libraries files to h:/ccm_wa/moreuse/sde-kazakov/sde/sde_template/comps/tmComp8/lib/x86_nt_el_i486
...
copying the DLL files to h:/ccm_wa/moreuse/sde-kazakov/sde/sde_template/comps/tmComp8/lib/x86_nt_el_i486/_sp_g
```

It is important to know that SDE2 copies the DLLs, if any, to the directory name determined from _TMDIVERSITY and _TMTGTREL.

3.10.1.2 Binary release for the object files

Generation of the binary release for object files is a typical BSL (board support library) issue and you, as a developer, will not need this unless you work with BSL.

SDE2 makes a binary release of the object files if _TMTGTCOPYOBJ is set to one or more of the produced object files, for example,

```
..\comps\tmComp1> set _TMTGTCOPYOBJ=tmComp1 tmComp4
```

If this variable intersects with one of the generated object files in the _SDE_OBJECTS makefile variable (i.e., if such file is produced), this file is copied in the binary release.

SDE2 prints the message:

```
copying the object files c:/b_result_1206/./comps/tmComp1/tmp/x86_nt_el_i486_g/src/tmComp1.obj to
h:/ccm_wa/moreuse/sde-kazakov/sde/sde_template/comps/tmComp1/lib/x86_nt_el_i486/_g
```

It is important to notice that this object file is copied into a subdirectory of the `lib` directory in the component. The name of this directory is often different than the name of the directory where the DLLs are produced. Here we will explain the reasons why it is done so. The object file compilation depends on **_TMTGTREL** and **_TMDIVERSITY**. You can always add new fields to **_TMDIVERSITY**, but the object file is sensitive only to a part of these fields. So, the object file is placed in the directory which name depends on **_TMTGTREL** and **_<CompName>_DIVERSITY** (extracted from **_TMDIVERSITY** in components **diversity.mk** file) only. And example for **comps/tmComp8** is:

```
_TMTGTREL=debug
_TMDIVERSITY=_mp_flo_
_tmComp8_DIVERSITY=_mp  # because tmComp8 knows only _mp_ and # _sp_ diversities.
```

Then `tmComp8/lib/<lib.config>/_mp_g` will be the directory name where the object file is placed.

In this way you can add the new values to **_TMDIVERSITY** and they do not reflect to the directory name where the object file is placed.

3.10.2 Using a binary release

3.10.2.1 Using a binary release for libraries and DLLs

You can use a binary release if you compile an executable/DLL, if such a binary release is present. If you require a library/DLL with your **LIBS** statement, for example,

```
LIBS=tmComp8
```

SDE2 always looks for such library/DLL

- First in the generated release directory (i.e., `<_TMTGTBUILDROOT>/comps/generated/lib/<your config>`)
- If not present, in the components' binary release

The only reason to give preference to the generated release directory is the fact that one could make some fixes in the component and then regenerate this component.

3.10.2.2 Using a binary release for object files

This usage is supported only for the executables. You can use a binary release for the object files if you set in your makefile:

```
REQUIRES += tmComp1
_SDE_OBJECTS += tmComp1.$(_SDE_O)
-include $(_tmComp1_DIR)/diversity.mk
```

SDE2 will find **tmComp1.$(_SDE_O)**, if present in the binary release, but you should also add **tmComp1** to the **REQUIRES** section and if **tmComp1/diversity.mk** is present, you have to add it manually to your makefile, because SDE2 needs the definition of **_tmComp1_DIVERSITY**, if it is defined. You can get the same effect if you have **tmComp1** in your **LIBS** section, but in this case SDE2 will require **tmComp1**'s library (although it will not be used).

3.10.2.3 Advantages of a binary release approach

The binary release gives big advantages in the organizational aspect and we will illustrate this with the following real example.

Let's assume that you use a project containing 50-100 components and you have 10 developers, each of them producing new components. So, all of those developers should use the same libraries/DLLs, but all of them should have their own release tree in order to test their own projects. You can make a binary release for the used components and put them, together with their binary release, on a network drive. The developers could use these components setting in their **<_TMROOT>/project/prjlist.txt** file its name (up to the **comps** directory), together with the name of its working project, for example,

n:/dvp_release/infra n:/dvp_release/kernal c:/my_prj

3.11 IDL support in SDE2

This section provides you with an overview of Interface Definition Language and describes the location and usage of the IDL files in the SDE2 distribution

3.11.1 Interface Definition Laungage (IDL)

IDL is language-neutral Interface Definition Language. It is created and supported by the OMG group (**www.omg.com**). The purpose of IDL is to define a general interface language for header files and to be able to map these files to the header files for different languages, such as C, C++, Java, Cobol, Ada, etc. The intention of IDL is to make communication between the objects written in different languages easier and to provide support for CORBA (Common Object Request Broker Architecture).

Next to OMG IDL, multiple (incompatible) dialects of IDL exist, most notably MIDL, from Microsoft. For NXPIDL, (a subset of) the MIDL dialect is used. This includes the use of various attributes, specified between square brackets, for example: **[uuid(...)]**.

Each compiler supplier could provide its own tool to transfer IDL files to header files and in some cases, to source files. For example, Sun Microsystems(c) provides an **idltojava** tool, see **java.sun.com**. NXP Semiconductors provides its own tool for handling IDL files called **nxpidl** both for NT, Unix and Linux platforms. The tool is distributed from the SoCDT/PID department

3.11.2 Location of the IDL files

The IDL files are located in the subdirectory **idl** of the corresponding interface or component directory.

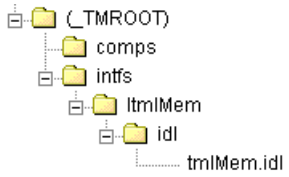


Figure 3-12: Location of IDL files

In the case when the IDL files are located in the component, they are used to describe the components that contain (one or more) coclasses that implement the IDL interfaces. These IDL files are located in the **comps** tree, not in the **intfs** tree.The co-class definition idl files will have the file extension ***.cdl**

In the case when the IDL files are located in the interface, they are used to describe the interface definitions. These IDL files are located in the **intfs** tree.The interface definition idl files will have the file extension ***.idl**

The **intfs** tree might also contain another type of idl file, used for type definitions. Thease IDL files will have the file extension ***.ddl**

The header and guid source files are generated if an IDL rule is present. The locations of the generated files is as follows:

For any interface definition idl file (*.idl)

<_TMTGTBUILDROOT>/intfs/generated/inc - for header files

<_TMTGTBUILDROOT>/intfs/generated/src - for guid c files

For any co-class definition idl file (*.cdl)

<_TMTGTBUILDROOT>/comps/generated/src - for guid c files

For any type definition idl file (*.ddl)

<_TMTGTBUILDROOT>/intfs/generated/inc - for header files

3.11.3 Usage

We are using a not-integrated approach. This means, that if you want to use the generated header files, you should do some extra action: either copy them manually in the **inc** interface directory, or set your makefiles to look at the location of generated header files. Further, if you want to generate those files from SDE2, you should add an IDL target in your targets list, for example,,

```
LOCAL_INCLUDES = $(_SDE_TMTGTBUILDROOT)/intfs/generated/inc
all: configuration idl lib
```

We support both 32- and 128-bit GUIDs (Global Unique IDentifier). By default we use 128-bit GUID. You can use a 32-bit GUID if you define _TMTGTGUIDS to 32. It will add the following defines to your compilation line

```
LOCAL_CFLAGS += -DTMFL_GUID_IS_32_BITS
LOCAL_CXXFLAGS += -DTMFL_GUID_IS_32_BITS
```

IIt is important to know that we generate all header files (except those specified in the next paragraph) based on the REQUIRES section of the makefile and local `idl` directory, for which IDL file is found. SDE2 calculates the transitive closure of REQUIRES section to claculate all required interfaces. If any of the interfaces among them is defined as `idl, SDE2 would generate the corresponding header files.`,

```
REQUIRES    =       dvp         \
                    dvpdebug    \
                    tmIUnknown  \
                    tmIClassFactory \
                    tmIMem      \
                    tmCom
```

During the generation of header files, there is a list of IDL files, which should not produce header files. Those IDL files are used only because the nxpidl compiler does not compile, if it does have them. This list is set in the _SDE_FILTEROUT_IDL makefile variable by default to tmIUnknown.idl tmNxTypes.idl tmComGuid.idl and tmMidlCompt.idl . You can redefine it, if you have other such IDL files.

Please check whether you have set in your project/sites/<_TMSITE>/<UNAME>.mk file:

IDLTOOL = nxpidl

for your platform (NT, Unix, Linux, Sun Solaris).

3.11.4 Binary Release of IDL-Guid files

SDE2 can generate the binary release of header files and Guid libraries, generated from `idl` files.

The user needs to set an environmental variable _TMTGTCOPYGUIDS to 1

The header files generated from both interface definition file (*.idl) and type definition file (*.ddl) would be copied on to `<_TMROOT>/intfs/<intfsName>/inc` directory.

The Guid library would be generated only if you are building either a dll or an executable. SDE2 encapsulates this library with the generated dlls or executables. However, if _TMTGTCOPYGUIDS is set to 1, this guid library (for example, guild_g.lib) would be copied to `<_TMROOT>/comps/<compName>/lib/<Lib Configuration>.`

3.12 Qmore invokation from SDE2

Qmore qualifies a software component for reusability as per the standards defined in MoReUse™. These standards are laid out, maintained and evolved by the System on Chip – Design Technology group within NXP-CTO (SoCDT).

Qmore performs the checks on the components as per the MoReUse™ standards and then certifies the component based on the results of these checks. As of this release, Qmore has a command line interface only.

The Qmore tool performs its check based on a MoReUse Metadata File (MMF). Attributes of the

CI and the checks to be performed on it can be specified in this file. Qmore follows this file for

performing all the checks. In short, the course of Qmore can be configured by way of the MMF.

Please refer to the Qmore user manual [QMORE] Qmore 03.01.00 User Manual, version 1.0, for more details on the MMF file and the Qmore tool.

The user can invoke Qmore tool, to run a compliance check on a particular component, group of component or a product (A group of logically related components). AN environment variable **QMORE_HOME** should be set to the location of the Qmore installation directory

For example, **QMORE_HOME = d:/qmore/bin**

The user needs to type **gmake qmore** to invoke qmore on any particular component. By default, Qmore would be run on that partticualr component, on which it is invoked. (Please note that the component under test should contain a MMF file - <CompName>.ccf)

Qmore can be run on all components used to build an alpplication by using the build_exe script as follows:

build_exe.pl apps/<ApplicationName> -target:qmore

We can also invoke Qmore on a set of components by specifying the environment variable

QMOREPARENTDIR. We need to set this environment variable with the location of the parent directory, which holds all the components under test.

For example, **QMOREPARENTDIR = d:/sde_template/comps**

We can also invoke Qmore on a product (A set of all related diredctory constituting that product) by specifying an environment variable QMOREPRODDIR. We need to set this environment variable with the location of the product directory.

For example, **QMOREPRODDIR = d:/infra**

Both these environment variables can have either relative path or obsolute path settings.

3.13 SDE2 Perl Scripts

3.13.1 Perl

The build scripts require Perl version 5.005_03 (Unix distribution) / 5.6.1 (Windows NT or 2000 or XP distribution) or later.

If necessary, Perl can be downloaded for Windows NT/2000/XP from
http://www.activestate.com/Products/ActivePerl/download.plex

SDE2 additionally provides the scripts in executable form for WinNT. According to the supplier, these executables should also work on Windows 98, 2000 and XP. They can be downloaded form the SDE2 website at:

http://pww.rtg.sc.philips.com/cmd/html/sde2_1_5.html

3.13.2 Build scripts overview

The SDE2 is supplied with three build scripts:

- **build.pl** – Used for building all relevant configurations of one or more components; it sets the relevant environment variables. It does not set platform-specific settings; they should be set in advance. A developer and/or tester will usually only use it to build one component. The "more components" application is meant for configuration managers. The use of the script is described in *Section 3.13.3, Build.pl* on page 67. For more details see the User's Manual of Build Component Script
- **build_exe.pl** – Used for building an executable and its required libraries for one specific configuration. This script is meant for the developers and testers of components. The use of this script is described in *Section 3.13.4, Build_exe.pl* on page 72.
- **requires.pl** – Computes the list of all components and interfaces that are needed to build a component. The use of this script is described in *Section 3.13.5, Requires.pl* on page 75.
- **auto_det.pl** – Used to check/validate the environment being set, to build the applications or components and also the makefiles of the components for any possible errors. The use of this script is described in *Section 3.13.6, Auto-detection script (auto_det.pl)* on page 75.
- **makefile_template.pl** – Used to generate a template makefile to be supplied to other customers to build the reusable componets/applications in their environment. The use of this script is described in *Section 3.13.7, Makefile Template Script (makefile_template.pl)* on page 76.

- **generate_diversity_mk.pl** – Used to generate diversity.mk file based on the entries in **configurations.txt** file. The use of this script is described in *Section 3.13.8, Script to generate diversity.mk file (generate_diversity_mk.pl)* on page 79
- **application_diversity.pl** – Used to generate all possible diversity values (including dependent components diversity information) that needs to be set in the environment to successfully build an application or a component based on the entries provided in **configurations.txt** file. The use of this script is described in *Section 3.13.9, Script to display the diversity information(application_diversity.pl)* on page 79
- **findtrailingspaces.pl** – Used to find out trailing spaces in makefiles in a component or in SDE2 makefiles. The use of this script is described in *Section 3.13.9, Script to display the diversity information(application_diversity.pl)* on page 79
- The scripts are located in the directory **`sde/scripts`**.

The first two build scripts are tools for building the components and executables. All builds can be performed without using the build scripts. The advantage of build scripts is:

- They change the directories, environment variables, invoking **make/gmake**
- **build_exe.pl** determines the correct order in which to do this
- They report everything in one (two) file(s)
- It is convenient for overnight builds, etc.

The scripts generate logging information in log files. In *Section 3.13.6, Auto-detection script (auto_det.pl)* on page 75, the log files are described in more detail. The scripts also print all the information to **stdout** and/or **stderr**.

3.13.3 Build.pl

3.13.3.1 Input of the script

The script needs the following information:

- Per component a configuration file.

  This is by default the **configurations.txt** file. See *Section 3.13.3.2, Component configuration files* on page 68 for more information on this file.

- The overall project configuration file

  This is the file **configurations.txt** that is available in the **`$(_TMPROJECT)`** directory. See *Section 3.13.3.3, The overall configurations.txt file* on page 70 for more information on this file. It is also possible to tell the script to use a specific configuration file (for example, a common project configuration file) by using the **-config** option (see *Section 3.13.3.4, Invoking build.pl* on page 71).

- A list of components to build.

  This is by default the **buildlist.txt** file that is available in the **`$(_TMPROJECT)`** directory. There are two ways to supply the list to the script:

  - Including the components explicitly on the command line (see *Section 3.13.3.4, Invoking build.pl* on page 71 for more details).
  - Telling the script to use an alternative ‘buildlist’ file by using the **-list** option (see *Invoking build.pl* for more details).

The **buildlist.txt** file contains per line the location of the main directory of a component. Below is an example of a **buildlist.txt** file for a project containing two components and an overall application:.

```
comps/tmApe
comps/tmNut
apps/overall
```

3.13.3.2 Component configuration files

Each component that is part of the build process must contain a **`configurations.txt`** file that is located in the main directory of the component. Example: for the component **`Comp1`** this directory is **`comps/tmComp1`**.

- The configuration file contains the locations of the makefiles of the component and per makefile
  The location of one makefile related to the location of the **configurations.txt** file.
- A set of valid configurations for that makefile.
- The location of (test) executable makefiles related to the location of the **configurations.txt** file.
- A set of valid configurations for each such (test) executable.

This is done in the following way:

This is implemented as shown below.

```
<location makefile 1>/makefile
configuration1_1
configuration1_2
...
EXECUTABLE
<location makefile 2>/makefile
configuration2_1
configuration2_2
...
EXECUTABLE
<location makefile 3>/makefile
configuration3_1
```

A configuration is specified as follows:

```
<_TMTGTCPUCLASS>~<_TMTGTOSCLASS>~<_TMTGTOS>~<_TMTGTENDIAN>~
<_TMTGTCPUTYPE>~<_TMDIVERSITY>~<_TMTGTREL>~<_TMTCSHOST>~<_TMBSL>~
<_TMLINKTYPE>~<_TMTOOLCHAIN>
```

In case there is no component makefile (only executables are present), the part before the `EXECUTABLE` line is not present.

The three fields (**_TMTCSHOST**, **_TMBSL**, **_TMLINKTYPE**) only apply for executables, not for the components. The field **_TMTOOLCHAIN** is optional, the default value is empty.

For an explanation of the fields, see *Section 3.2.1, Selecting a configuration* on page 17. The _TMTCSHOST field is only present for the tm-psos configuration class. The _TMDIVERSITY field is discussed in *Section 3.5.1.1, Extend the makefile with a _TMDIVERSITY selection* on page 37. The last three fields are optional. They are only relevant for executables. The _TMBSL and the _TMDIVERSITY fields may be empty.

Note: In case the _TMDIVERSITY field in the configuration file is empty, the script uses the value of the environment variable of the same name (which may also be empty). We recommend you leave this field empty in your makefiles. One cannot predict what kind of diversity product's user will define for a component. If the platform does not generate DLLs this is not a problem. However, all the DLLs end up in one directory and its name depends on _TMDIVERSITY. So, we cannot change _TMDIVERSITY during the (build.pl) build process, otherwise the build can fail.

In the current release of SDE2, _TMLINKTYPE does not play a role in the compilation process except for tm_psos configuration. It can be set to static and dynamic without any difference for the output files. The DLLs' generation depends on the EXPORTS variable, see *Section 3.6, Dynamic link libraries (DLLs)* on page 42. The link type for TriMedia is set to be dynamic if there is at least one DLL. The standard link types for Microsoft compilers are used. The EXETYPE:DYNAMIC option for them is used when building a virtual device driver (VxD), so there is nothing to do with _TMLINKTYPE field. It can be set as a separate option.

The following fields may be substituted by a wildcard (*):

- _TMTGTENDIAN
- _TMTGTREL
- _TMTCSHOST
- _TMLINKTYPE
- _TMBSL

In this way one line can specify multiple configurations.

Note that the configuration file can contain white lines and comment lines. Comment lines start with a #. The file can contain space, tab and \^M (control-M) symbols.

Example of a `comps/tmosal/configurations.txt` (`_TMTOOLCHAIN` is not present, the default empty value is accepted):

```
makefile
tm          ~psos~psos200~* ~tm32~_multi_~*
tm          ~psos~psos200~* ~tm32~_single_~*
mips        ~psos~psos250~eb~r3940~debug
EXECUTABLE
tst/Tst1/makefile
tm          ~psos~psos200~* ~tm32 ~                                ~*                  ~nohost    ~ ~*
EXECUTABLE
tst/Tst2/makefile
mips        ~psos~psos250~eb~r3940 ~                               ~debug
~ _p4032~static
```

The first line indicates that there is a makefile at location **$(_TMROOT)/comps/tmosal.** This makefile can be built for some **tm-psos** configurations with **_TMDIVERSITY=_multi_.** It can be built for some tm-psos configurations with **_TMDIVERSITY=_single_.** It also can be built for exactly one **mips-psos** configuration.

The second makefile is at location **comps/tmosal/tst/Tst1** and can be built for some **tm-psos** configurations with **_TMDIVERSITY** is empty. The last makefile is located at **comps/tmosal/tst/Tst2** and can be built for exactly one mips-psos configuration with **_TMDIVERSITY** is empty.

3.13.3.3 The overall configurations.txt file

It can be the case that a development project imports a component that is buildable for a Trimedia as well as a mips environment, while the project is only interested in a Mips environment. Each development project has got a *configurations.txt* file that describes what configurations are relevant for the project. The file is located by default in **$(_TMPROJECT)**. However you can specify the project configuration file via the command line. You have to specify **-config:<FilePath/FileName>** in the build.pl command line.

Each line of the configuration file contains a configuration. A configuration equals the configuration as described in the previous section (3.11.3.2) except that the **_TMDIVERSITY** field is not there.

Example of a project *configurations.txt* file:

```
tm ~psos~psos200~* ~tm32 ~*    ~nohost~~*
mips~psos~pso250 ~eb~r3940~debug~_p4032 ~ static
mips~ce ~ce212 ~el~r3000~*    ~vpci1 ~ *

.....
```

3.13.3.4 Invoking build.pl

If all environment variables from *Table 3-2 Additional environment variables per configuration class* and *Table 3-5 The generic environment variables* are properly set, the script can be invoked in the following way:

```
perl build.pl [-c] [-d] [-h] [-dbg] [-target:<target name>]
[-list:<Path/ListFileName>]
[-config:<Path/ConfFileName>] <C1> <C2> ...
```

The optional **-c** (short for continue) denotes that the script will proceed when an error is encountered. Default behavior is, that the script halts when an error is encountered.

The buildables can be specified in one of the following ways:

- **-list:<Path/ListFileName>** – Build all components listed in file **Path/ListFileName** using the default project configuration file **<_TMPROJECT>/configurations.txt**.
- **-config:<Path/ConfFileName>** – Build all components listed in the default buildlist file **<_TMPROJECT>/buildlist.txt** using the project configuration file **Path/ListFileName**
- **-d** – Suppress the DLL build. DLLs are not generated.
- **-dbg** – Prints to the standard output scripts' debug information during the execution process.
- **-target:<target name>** – Execute not default target for each of the components, example **clean** target.
- **<C1> <C2>....** – Build the components **<C1>**, **<C2>** etc.
- **-h** – Prints help information and exit.

Examples of how to execute a build:

```
perl build.pl -list:c:/ccm_wa/prjcomps.txt
perl build.pl –c -config:c:/cfg_x86.txt
perl build.pl -d comps/tmosal comps/tmml
```

3.13.3.5 Output of the script

The results of the build script are reported on 'standard out' along with the elapsed time (time taken to build each component) and also written onto two files: **build_script.log** and **build_script_report.log** in the **_TMREPORTS** directory.

- **build_script.log** contains all the information about the build process.
- **build_script_report.log** contains a summary of the results and list the build errors, if any.

There are three possible build results for each platform/configuration:

- Built successfully
- Built with warnings
- Build failed

NOTE: When using build.pl for an application that has dependencies on multiple components/libraries make sure that all the required componets are built for required flavors before building the application. buildlist.txt should be updated carefully based on this dependencies.

3.13.4 Build_exe.pl

The script **build_exe.pl** builds multiple executables and its required libraries. It requires that the relevant environment variables from Table 3-1, Table 3-2 and Table 3-5 have been set in advance.

The script is invoked as shown below:

perl build_exe.pl [-b] [-c] [-dbg] [-g|g: <filename>] [-h] [-l] [-list:<file name>] [-quiet] [-showtime] [-target:<target name>] <location of executable-makefile>

- **-b** – forces a build, even if a binary release is present (optional)
- **-c** – the script will proceed when an error is encountered (optional)
- **-dbg** – prints to the standard output script's debug information during the execution process
- **-g|g:<filename>** – prints the dependency graph information (optional) on STDOUT or <filename>
- **-h** – prints a help information and stops (optional)
- **-l** – generates only ***.l** files and stops (optional)
- **-list <filename>** – File that contains the list of applications to be built
- **-log** – Prints the information onto the report logfile (optional)
- **-quiet** – Prints a brief information of the build(optional)
- **-showtime** – Prints the table of time taken to build all the components individually and also the total time taken to build the application and its dependants (optional)
- **-target:<target name>** – executes the target **<target name>** for each of the components, transitively closed with **LIBS** and **JAVA_REQUIRES** dependency. For example, **−target:clean** will clean all libraries and intermediate files required from an executable/component. (optional)

Invoking the script results in building of all the executables and its required libraries as specified in the makefile. It will be built for the current configuration. For example,

perl build_exe.pl -c -b comps/tmComp1/tst/Tst1
perl build_exe.pl -c -b comps/tmComp1/tst/Tst1 apps/helloworld
perl build_exe.pl -list appslist.txt

3.13.4.1 Two iteration process with build_exe.pl

As described before, **build_exe.pl** uses the current settings of environment variables such as **_TMTGTOS**, **_TMTGTREL**, **_TMDIVERSITY**, ... and it does not change the values of these variables during the build process. It is doing two passes. In its first pass, it builds recursively ***.l** files, see Section 3.7.2, *DependsOn relation*. These files give information about:

- Required libraries which have to be built; all libraries are determined using the transitive closure of **LIBS** and **JAVA_REQUIRES** section, see Section 3.7, *Libraries and the LIBS section*.
- Required build modes (debug/assert/retail/trace). These modes may be different from the current build mode set in **_TMTGTREL**.

The script analyzes the information and computes the correct build order. That means if component A requires component B, component B is built before component A.

If you are building only a static library, you do not need this script, because the build process does not require presence of other libraries. However, if this is the case, **build_exe.pl** prints a warning message and builds all libraries determined using the transitive closure of the **LIBS** section. This is done in order to keep the behavior of the script consistent for executables and components.

In the second pass the script is doing the real build, based on the build order. If the **-b** option is not specified, it first checks whether there exists a binary release of the static library with the appropriate name. If so, the build process is skipped. In all other cases, the build process is started. If the build script discovers a discrepancy between the default build mode and the required build mode, it executes one of the extra goals **_force_debug**, **_force_assert**, **_force_trace** or **_force_retail**. More about this is discussed in the next section.

3.13.4.2 Mixing debug/assert/retail/trace diversities and build_exe.pl

When developing an application, it is very helpful to use some set of libraries in a retail mode and others in a debug mode. This section describes a way to do this. The underlying issues are similar to those described in *Section 3.7.3, Overriding default diversities and *.l files* on page 47.

In static builds, diversities are identified using suffixes (**_g**, **_a**, etc.). This method works for static builds. The situation is more complex for DLLs, and that is not covered here.

The application is responsible for specifying the mixture of suffixes. This information is given in the application's makefile. For every library specified in the **libs** section, the application makefile can also have a matching "suffix" entry.

For example: Build the application in debug mode. This allows for debugging parts of the applications that need debugging. Use the "suffix" command to specify that most libraries are to be used in "assert" mode so as to minimize the impact of the debugging on the overall system.

In this example makefile excerpt, the application is designed to be built in debug mode with a mixture of library flavors.

```
#-----------------------------------------------------------------
# Required components
#-----------------------------------------------------------------

REQUIRES      =
\
   tmOsal
\
   tmUtil
```

```
\
   tmMl
\
   tmbslCore
\
   tmbslBoards
\
   tmDbg
\

LIBS
=                                                \
   tmDbg
\
   tmSpOsal
\
   tmUtil
\
   tmMl
\
   tmbslCore
\
   tmbslBoards
\

_tmDbg_SUFFIX = _g          # use the debug version of this one
_tmSpOsal_SUFFIX = _a       # use the assert version
_tmutil_SUFFIX =            # use the retail version
_tmml_SUFFIX = _g           # use the debug version of this one
_tmbslCore_SUFFIX = _a      # use the assert version
_tmbslBoards_SUFFIX = _a    # use the assert version
```


3.13.4.3 Output of the script

The results of the build script are reported on 'standard out' along with the elapsed time (time taken to build each component), also written onto the file: **build_exe_report.log** in the subdirectory of **_TMREPORTS** directory, if the **-log** command-line option is used. The name of the subdirectory is determined by the configuration and current date and time. In this order, you can be sure that if you run multiple scripts, they all will have their output files in a different directories.

- **build_exe_report.log** contains a summary of the results and list the build errors, if any.

There are two possible build results for each component:

- Built completed
- Build failed

3.13.5 Requires.pl

The script **requires.pl** generates a list of components and interfaces that have to be built before a specific component can be built. Here, instead of the transitive closure of the **LIBS** relation as in the previous chapter, we are looking for the transitive closure of the **REQUIRES** and **JAVA_REQUIRES** relation. The idea behind this script is that you can use one component only if you have all the required components. So, if you get a component, you should get all the required components and interfaces.

The script is invoked in the following way:

perl requires.pl <components root directory>

For example,

```
>perl requires.pl comps/tmComp15/tst/Tst1 comps/tmComp3
comps/tmComp15/tst/Tst1::comps/tmComp15 comps/tmComp1 comps/tmComp2
comps/tmComp3:: intfs/ItmReal comps/tmRealFloat
```

The **::** is used as a separator between the component name and all required components.

3.13.6 Auto-detection script (auto_det.pl)

This script tests the settings of required environment variables and component makefiles of SDE2. This script is capable of testing both the environment and the component makefiles separately or together, depending on the option selected.

This script is designed for all SDE2 users and it helps you in identifying any deviations in the settings of environment and component makefiles, for a successful build.

This script allows you to run only a first-level check. It is recommended you run this script before sending queries/questions to SDE2 team.

The auto-detection script is invoked as follows:

perl auto_det.pl -env|-comp:(<CompName>/all)|[-showall]|-h

- **-env** option allows the script to test the environment variables.
- **-comp:<CompName>** option makes the script to check only the component makefile variables for the specified component name in **<CompName>. <CompName>** should always start with **comps** directory. For example: **comps/tmComp1**, **comps/tmReal** etc.
- **-comp:all** option makes the script to check the component makefile variables of all the components specified in **loc_list.*** files.
- **[-help]** option prints help/usage and version information.
- **[-showall]** option prints detailed messages during the execution of the script.

Examples of how to execute the script:

```
perl auto_det.pl  -env
perl auto_det.pl  -comp:comps/tmComp1 -env
perl auto_det.pl  -comp:all -env -showall
perl auto_det.pl  -h
```

3.13.7 Makefile Template Script (makefile_template.pl)

This script **(makefile_template.pl)** produces makefile template, for editing application makefiles, to build applications outside SDE2. The makefile template is produced, for the current configuration, set in the environment. This script produces a template makefile in the **_TMREPORTS/<configuration>** directory.

Currently the following configurations are supported:

x86_nt, tm_psos, x86_ce, arm_ce, mips_ce, x86_vxworks, arm_vxworks, arm_nullos, hp_nullos.

The makefile template script is invoked as follows:

perl makefile_template.pl [-h]

- **-h** – prints the help message.
- generates the makefile template without any options.

The sample below provides an example of a **makefile** generated in _TMREPORTS/x86_nt directory for x86_nt configuration.

```
#########################################################################
#              Makefile template for building application outside SDE2      #
#########################################################################
#-------------------------------------------------------------------
#This is a makefile template, that can be used to build an application #out side SDE2 environment. The user can create his
own directory #structure to place the required source files, header files, libraries #and build tools. Those directory
locations need to be specified in the #appropriate locations in this makefile template, to make this a complete #wokable
makefile
# -------------------------------------------------------------------------------------------------------------------------
# Enter the location information here. Please do not add any extra space #at the end of line
#-------------------------------------------------------------------
#ROOT_LOCATION --> Location of the root directory
#SRC_LOCATION  --> Location of source files
#INC_LOCATION  --> Location of header files
#LIBS_LOCATION --> Location of library files
#OBJ_LOCATION  --> Location of object files
#EXE_LOCATION  --> Location of executable generated
# -------------------------------------------------------------------
ROOT_LOCATION =
SRC_LOCATION  =
INC_LOCATION  =
LIBS_LOCATION =
OBJ_LOCATION =
EXE_LOCATION =
# -------------------------------------------------------------------
# Enter the source file information here.Please do not add any extra space at the end of line
# -------------------------------------------------------------------
C_SOURCES     = $(SRC_LOCATION)/<C sourcefile>
C_FILES       = $(notdir $($(C_SOURCES))
CXX_SOURCES   = $(SRC_LOCATION)/<C++ source file>
CXX_FILES     = $(notdir $($(CXX_SOURCES))
S_SOURCES     = $(SRC_LOCATION)/<Assembly source files>
S_FILES       = $(notdir $($(S_SOURCES))
#-------------------------------------------------------------------------------
# Definition of object files
#-------------------------------------------------------------------------------
OBJECT_FILES += $(CXX_FILES:%.cpp=$(OBJ_LOCATION)/%.obj)        \
                                        $(C_FILES:%.c=$(OBJ_LOCATION)/%.obj)            \
                                        $(filter %.obj,$(S_FILES:%.s=$(OBJ_LOCATION)/%.obj)\
                                        $(S_FILES:%.S=$(OBJ_LOCATION)/%.obj)
                                        $(S_FILES:%.asm=$(OBJ_LOCATION)/%.obj))
```

```
#-------------------------------------------------------------
# Definition of Library file
#-------------------------------------------------------------
LIBRARY_FILE = $(LIBS_LOCATION)/<Enter the library name>.lib
#-------------------------------------------------------------
# Definition of executable file
#-------------------------------------------------------------
EXE_FILE = $(EXE_LOCATION)/<Enter the executable name>.exe
#-------------------------------------------------------------
# Definition of Compiler options
#-------------------------------------------------------------
sde_copts := -DTMFL_REL=TMFL_REL_DEBUG -Zi -DDEBUG -Od -MDd -W4 -DWIN32 -D_WIN32 -D_WINDOWS
-D_X86_ -D_MBCS -D_USRDLL -DX86_CPU -D_M_IX86=1 -I. -I$(SRC_LOCATION) -I$(INC_LOCATION) -I<Add
other include Paths Here> -GX -Zm1000 /nologo
sde_cxxopts := -DTMFL_REL=TMFL_REL_DEBUG -Zi -DDEBUG -Od -MDd -W4 -DWIN32 -D_WIN32
-D_WINDOWS -D_X86_ -D_MBCS -D_USRDLL -DX86_CPU -D_M_IX86=1 -I. -I$(SRC_LOCATION)
-I$(INC_LOCATION) -I<Add other include Paths Here> -GX -Zm1000 /nologo
sde_aopts :=
#-------------------------------------------------------------
# Definition of Linker options
#-------------------------------------------------------------
sde_ldopts := /nologo -DEBUG /WARN:3 /LIBPATH:d:/progra~1/micros~1/VC98/lib
VPATH = $(SRC_LOCATION) $(OBJ_LOCATION) $(LIBS_LOCATION)
#-------------------------------------------------------------
# Compiler/Linker toolchains
#-------------------------------------------------------------
CC          = cl.exe
CXX          = cl.exe
IDL         =
LB          = lib.exe
AS          =
LD          = link.exe
#-------------------------------------------------------------
# Rules for generating object files
#-------------------------------------------------------------
$(OBJ_LOCATION)/%.obj: %.c
            mkdir -p $(dir $@);   \
            echo compiling $<;              \
            $(CC) $(sde_copts) -c -Fo$@ $<
$(OBJ_LOCATION)/%.obj: %.cpp
            mkdir -p $(dir $@);   \
            echo compiling $<;              \
            $(CXX) $(sde_cxxopts) -c -Fo$@ $<
```

```
$(OBJ_LOCATION)/%.obj: %.s
        mkdir -p $(dir $@);  \
        echo compiling $<;          \
        $(AS) $(sde_aopts) -c -Fo$@ $<
$(OBJ_LOCATION)/%.obj: %.asm
        mkdir -p $(dir $@);  \
        echo compiling $<;          \
        $(AS) $(sde_aopts) -c -Fo$@ $<
#-----------------------------------------------------------
# Rules for generating Library files
#-----------------------------------------------------------
$(LIBRARY_FILE): $(OBJECT_FILES)
        mkdir -p $(LIBS_LOCATION);  \
        echo building library $@;          \
        $(LB) /nologo /verbose /out:$@ $(OBJECT_FILES)
#-----------------------------------------------------------
# Rules for generating Executable
#-----------------------------------------------------------
$(EXE_FILE): $(LIBRARY_FILE) $(OBJECT_FILES)
        mkdir -p $(EXE_LOCATION);  \
        echo building executable $@;          \
        $(LD) $(OBJECT_FILES) <Other Object Files> -LIBPATH:$(LIBS_LOCATION) <add other libs location if
any> $(LIBRARY_FILE) <Add other DLLs and Libraries in correct order> \
                $(sde_ldopts) /out:$@
#-----------------------------------------------------------------
# Define the required target that you need to build
# Choose either $(LIBRARY_FILE) or $(EXE_FILE) for building library or executable respectively
#-----------------------------------------------------------------
all: $(OBJECT_FILES) $(LIBRARY_FILE) $(EXE_FILE)
```

3.13.8 Script to generate diversity.mk file (generate_diversity_mk.pl)

This script **(generate_diversity_mk.pl)** generates **diversity.mk** file of a component based on the **configurations.txt** file located in the same component.

This script is invoked as follows:

cd <component directory>
perl $_TMROOT/sde/scripts/generate_diversity_mk.pl

- Generates the diversity.mk file based on the component's configurations.txt file.

3.13.9 Script to display the diversity information(application_diversity.pl)

This script **(application_diversity.pl)** generates a list of all possible combinations of diversity required to build an application based on the all the dependent components' **configurations.txt** files.

This script is invoked as follows:

cd <application/component directory>
perl $_TMROOT/sde/scripts/application_diversity.pl

- Displays the possible combinations of diversity that needs to be set based on all the dependent components' configurations.txt files

3.13.10 Script to find trailing white spaces (findtrailingspaces.pl)

This script **findtrailingspaces.pl** detects leading white spaces in a makefile mentioned in the command line. If no files are specified in the commandline then the script checks all the SDE2 makefiles.

This script is invoked as follows:

perl $_TMROOT/sde/scripts/findtrailingspaces.pl -f makefile

- Checks makefile in the current working directory for possible trailing white spaces.

perl $_TMROOT/sde/scripts/findtrailingspaces.pl

- Checks all the *.mk files in the SDE2 installation directory.

3.13.11 Setting environment variables of SDE2(setenv.bat)

This is top level batch file that creates batch file containing configuration specific environment variables, runs the batch file and sets environment variables for that configuration to run SDE2. It also verifies the settings of the required environment variables,components and associated makefiles of SDE2 by executing auto-detection(auto_det.pl) perl script.

Setenv.bat file is invoked as below:

```
>setenv.bat
```

The batch file calls the following perl scripts:

- EnvCreate.pl
- Runs the batch file and sets environment variables for SDE2
- AutodetExecute.pl

3.13.11.1 EnvCreate.pl

This script creates the configuration specific batch file at the following location.

<SDE2 installation Directory>\<SDE2 Root Directory>\project\sites\<site name>

For example:

c:\sde2_23\sde_template\project\sites\blrsdm

The name of batch file created is as below

<_TMTGTCPUCLASS><_TMTOOLCHAIN}>_<_TMTGTOSCLASS>_<_TMTGTREL>_<_TMLINKTYPE>_<_TMDIVERSITY>_<_TMTGTENDIAN>.bat

It invokes graphical user interface to set the general environment variables as mentioned in *Table 3-5 The generic environment variables*.

Another graphical user interface is called to set the specific environment variables for the configuration class selected as mentioned in *Table 3-2 Additional environment variables per configuration class*.

At present, the configuration classes as mentioned in Configuration Class column of Table 3-2 are supported.For the configuration classes not mentiond in Table 3-2, only the generic environment variables are written in the batch file.

3.13.11.2 AutodetExecute.pl

This script calls **`auto_det.pl`**. It thus checks the presence and settings of the environment variables and component makefile variables.

3.14 Build flavors

In this chapter different minor so-called flavors in SDE2 are explained.

3.14.1 Dependency generation

In SDE2 dependencies are automatically generated. Changes to the files are tracked by means of time stamps. Dependency checking can be switched off by setting the environment variable **_TMNODEPENDENCIES** to 1. Omitting dependency checking results in shorter compilation time. The disadvantage is that it will not regenerate your libraries if one of required header files is updated.

For all configuration and source files (***.c**, ***.cpp**, ***.s**, ***.S**, ***.asm**) the corresponding file with dependency information is generated unless the environment variable **_TMNODEPENDENCIES** is set to 1. The name of the generated file is the same as the name of the corresponding source file, but with extension **d** instead of **c**, **cpp** etc.

Note: Working with dependencies and configuration management tool might be ambiguous. If you delete a new version of a file, this file will be replaced with an older version with an older date. So, the corresponding library will not be updated if you have generated ***.d** files.

Note: It is not allowed in SDE2 to have source files with the same name and different extension in the same directory in one component.

Per source file **<NAME>.<EXT>** the following file is generated:

<_TMTGTBUILDROOT>/comps/<component>/tmp/<_TMTGTCPUCLASS>_<_TMTGTOSCLASS>_<_TMTGTENDIAN>_<_TMTGTCPUTYPE>/src/<NAME>.d

For executables the following file is generated:

<_TMTGTBUILDROOT>/comps/<component>/tst/<tst dir>/tmp/<_TMTGTCPUCLASS>_<_TMTGTOSCLASS>_<_TMTGTENDIAN>_<_TMTGTCPUTYPE>/src/<NAME>.d

This file contains the list of all files on which the source file depends.

3.14.2 Copying objects into the release directory

SDE2 provides a target for copying (object) files into the release directory. The full names of the object files have to be specified into the **_SDE_COPYLIST** and the **_sde_copy_objects** target has to be invoked. Example (in the makefile):

```
_SDE_COPYLIST=$(DIR_INTERM)/src/tmComp1.$(_SDE_O)
all: configuration lib _sde_copy_objects
```

3.14.3 Build diagnostics

While building, extra build information is available by setting the environment variable **_TMECHO** to 1.

3.14.4 Memory image build support

SDE2 supports a memory build image for TriMedia. The image is built instead of a ***.out** file if both **_TMSTARTADDR** and **_TMENDADDR** are defined correctly. Additionally you can define:

- **_TMMMIOBASE** – Base address of MMIO. If it is undefined, a value of **0x1be00000** is used by default.
- **_TMCLOCKFREQ** – TriMedia clock frequency. If it is undefined, a value of **100000000** is used by default.

3.14.5 Debugging with SDE2

The **_sde_print_var** new target is introduced as follows.

```
_sde_print_var:
  $(_SDE_ECHO)$(ECHO) "$(var)=$($(var))"
```

It can be used for SDE2 debugging purposes, for example

make var=DIR_INTERM _sde_print_var

prints the value of **DIR_INTERM** and stops.

3.14.6 Debugging tools and SDE2

Some debugging tools, like Microsoft debuggers and the TriMedia debugger, need additional information to locate where the source files are. If this information is not available, you have to point each time to the location where source code is. This is inconvenient and therefore SDE2 provides a solution to make this process easier by using the absolute path of the source files rather than their relative path.

3.14.7 Using source files outside the component

In some cases a component file uses a source file that is located outside the directory scope of the component. An example is an interface with a piece of test code. This test code is used by the test code of a component implementing this interface. By extending the **VPATH** in your makefile you can deal with this situation. By definition **VPATH** specifies a search list of directories that **make** (**gmake**) should search for all files, including files which are targets of rules.

Below is a snapshot of a makefile of a test executable using a source file that is at another location (the **intfs** directory)

```
C_SOURCES     = \
        src/main.c \
        Itm2dPixmapFact/src/tm2dTestThisInterface.c
```

3.14.8 User-defined variables

SDE2 supports user-defined variables. They can be used in SDE2 and it is guaranteed that they will not be touched in future SDE2 releases or coincide with other user-defined variables.

The variable format is

SDEUD<Your component name>_<What ever the user has defined here>.

SDE2 also guarantees that it will not use **_TM_USER_DEF_<What ever the user has defined here>** in SDE2 makefiles.

3.14.9 Third-party software

Some software components or executables may require third-party software. SDE2 contains primitives to introduce additional search paths for third-party header files and linking third-party libraries from arbitrary locations.

Additional search paths for header files can be specified in makefiles (of components or executables) by modifying the line:

DIR_INCLUDE =

into

DIR_INCLUDE = <path1> <path2> ...

For example,

DIR_INCLUDE = $(MY_PATH1)/inc $(MY_PATH2)/inc

Additional libraries can be specified in makefiles of executables by adding the line:

EXTERNAL_LIBS = <path/lib1> <path/lib2> ...

For example,

EXTERNAL_LIBS = $(MY_PATH1)/lib/lib1 $(MY_PATH2)/lib/lib2

The **EXTERNAL_DLLS** are explained in *Section 3.6, Dynamic link libraries (DLLs)* on page 42.

The example for **EXTERNAL_LIBS** is given in **`tmComp10/tst/Tst1`**. As an external component **Comp1** is used, which is artificial (because **Comp1** is an internal component), but it is useful for test purposes.

Note that components that require non-SDE2–compliant software are less reusable. Avoid links to external software as much as possible. Also note that SDE2 adds the library extension to the external library names.

3.14.10 Use of inline qualifiers

The SDE2 CCB has decided not to use **inline** qualifiers in public header files. However they can be used in private header files and source files. The problems arise when one uses **inline** in interface files. Then the implementations of these functions are an integral part of the interface description. Therefore, it is impossible for a developer to define his/her own implementation of the interface.

3.14.11 Other targets

SDE2 supports other targets:

- **clean** – Calls **clean_lib** or **clean_target** depending on whether **makelib.mk** or **maketarget.mk** is invoked in the component's makefile.
- **clean_lib** – Cleans the corresponding libraries and their temporary files.
- **clean_target** – Cleans the corresponding executables and their temporary files.
- **clean_all** – Removes files **<_TMROOT>/project/loc_list.*** and removes the **_TMTGTBUILDROOT** directory if it is not empty. If **_TMTGTBUILDROOT** is empty, the following directories are removed: **<_TMROOT>/comps/generated** and **<_TMROOT>/<DIR_LOCAL>/tmp**
- **help** – Prints help information.

3.14.12 SDE2 warning messages

SDE2 checks for errors and prints the warning message if these errors are present. You can also put your custom warning message in the variable **_SDE_WARNINGS**.

SDE2 prints the following warning messages in the respective cases:

Warning: The interfaces $(_SDE_MISSING_INTERFACES) are missing in the project tree

Warning: $(notdir $(DIR_LOCAL)) is removed from the LIBS section, as it is a self reference

Warning: Circular dependencies detected! Check LIBS section of <related CompNames> components

Warning: <CompName>.l files is/are missing. Please read Chapter 3.7.4 of the User Manual!!

Warning: <CompName>.j files is/are missing. Please read Chapter 3.7.4 of the User Manual!!

3.14.12.1 Circular dependencies

If there is a circular dependency graph, SDE2 prints an warning. The dependency graph is determined from the **LIBS** relation in your makefiles. Even if your component refers to itself, it is an error. The reason for this is that it is not possible to make the correct library line for the linking of the final executable. The circular dependency arises, for example, if component A needs the interface and implementation of B's interface, and component B needs the interface and implementation of A's interface.

3.14.12.2 The loc_list files

The **loc_list.txt** and **loc_list.mk** files are generated from **prjlist.txt** files. They are updated if you touch the **prjlist.txt** file. SDE2 reports a warning if **_<CompName>_DIR** is not defined/found, i.e., something is wrong/not updated. The **loc_list** files are updated if you add a new component, if you touch your **prjlist.txt**, or if your definition of **_<CompName>_DIR** is invalid.

3.15 Component makefile manual

This section is a reference manual for a component makefile. It includes description of the recommended component makefile structure, the list of makefile variables defined with +=, tables with all makefile and environment variables used in SDE2, and a short description of all example components.

3.15.1 Structure

The makefile consists of several parts. Some of these are mandatory, some are optional. The order of the parts is also mandatory in some cases and can be varied in other cases. It is strongly advised to use the same order for the different parts as described in this manual in all cases.

This structure includes only the basic settings for Java compilation. You can find the complete list of all settings in *Appendix A*.

3.15.1.1 Component name and include environment.mk

This is the first part. The part is mandatory:

```
DIR_LOCAL= comps/tmComp1
include $(_TMROOT)/sde/environment.mk
```

Here you have to set the value of **DIR_LOCAL** in your component's directory.

You must not change the **include** line.

3.15.1.2 C, C++ and ASM source files

This part contains at least one reference to C, C++ or ASM file(s). It is as follows:

```
#-------------------------------------------------------------
# Source environment variables
#-------------------------------------------------------------
CXX_SOURCES =
C_SOURCES = src/tmComp1.c
CFG_SOURCES = cfg/tmComp1Cfg.c
S_SOURCES = src/$(PLATFORM_SPECIFIC_DIR)/tmComp1.s
```

Here you have to list your C++ sources in **CXX_SOURCES** and your C sources in **C_SOURCES**. The configuration sources (located in your component's **cfg** directory) are in **CFG_SOURCES** and the assembler files are in **S_SOURCES**. It is possible to use some custom environmental variables to set the directory path.

The path description is relative. It starts from **<_TMROOT>/<DIR_LOCAL>**.

It is not permitted to use source files outside the component directory structure. The exceptions are for executables. You can access a platform-specific directory of **<_TMROOT>/inc** directory, for example **drv_conf.c** is located in **<_TMROOT>/inc/mips_psos.** You can access other configurational source files using the **VPATH** makefile variable without the component scope.

3.15.1.3 REQUIRES section

This section is optional. It is required if your component requires other components. The part is as follows:

```
#-----------------------------------------------------------------
# Required components
#-----------------------------------------------------------------
REQUIRES      = tmComp2 \
            tmReal PROVIDED_BY tmRealFloat \
            tmComp8
```

You have to define a list of required components. The component names are separated by space(s). In case you need an interface (example: **tmReal**) implemented by a component (example: **tmRealFloat**), write this as follows:

<Interface Name> PROVIDED_BY <Component Name>

Putting a component in the **REQUIRES** list causes the public header files of that component to be in the include path.

If you implement an interface in your component you have to use the same construction as above.

Note that the interface subdirectory name in the **`intfs`** directory begins with **I**. This **I** cannot be present in the makefile, for example: **`intfs/ItmReal`**.

3.15.1.4 Recursive closure of REQUIRES section

An option is available in SDE2 to recursively close the REQUIRES section of a component or application. This can be done by setting the environment variable **_TMNESTEDINCLUDE** to 1 or any value

3.15.1.5 Third-party software and non default include directories

This section is optional. If necessary, put the include paths of the third parties or non default include paths in **DIR_INCLUDE**. Be aware, that third-party software will make your components less reusable. You can also use **DIR_INCLUDE** to access your private header files in a subdirectory of the **`src`** directory. Example:

DIR_INCLUDE = src/sub

3.15.1.6 Libraries and DLLs

This section is needed if you use functions from another component. Define it using the variable **LIBS**[6]. If you need an external library or DLL, set the complete path in **EXTERNAL_LIBS** and **EXTERNAL_DLLS**.

```
#-----------------------------------------------------------------
# Required libraries
#-----------------------------------------------------------------
LIBS      = tmComp1 tmComp6
```

6. In previous releases of SDE2, the variable **DLLS** was used to define required DLLs. This variable is not used any more since release 1.1.6.

```
#-----------------------------------------------------------------
# Required external libraries. Specify complete paths.
#-----------------------------------------------------------------
EXTERNAL_LIBS =
EXTERNAL_DLLS =
```

The LIBS contains list of the libraries separated by space(s).

Adding components to the LIBS section means you introduce a dependency between these two components. This dependency is important during the executable link time and it means that the required library will be placed after the requiring library in the compilation line. For more information about this, read *Section 3.7, Libraries and the LIBS section* on page 45.

Before SDE2 release 1.1.6, the default REL_SUFFIX had to be added at the end of the library name; from release 1.1.6 on, this is added automatically by SDE2. To create a library with a different compilation mode and/or different flavors, refer to *Section 3.13.4.2, Mixing debug/assert/retail/trace diversities and build_exe.pl* on page 73.

For the EXTERNAL_LIBS read *Section 3.14.9, Third-party software* on page 83.

For the calling style of DLLs line read *Section 3.6, Dynamic link libraries (DLLs)* on page 42.

3.15.1.7 EXPORTS variable

If the EXPORTS variable is set, the target is `lib`, and you have one of the following platforms: arm_ce, armgnu_linux, mips_ce, mipsgnu_linux , tm_psos, x86_nt, x86_ce x86gnu_linux or x86gnu_nullos, the compiler starts to generate a DLL after the generation of the library. Example for EXPORTS:

```
EXPORTS = tmComp8Print
```

3.15.1.8 Setting the diversity

This part is necessary if you have diversities in your component. This part must be appear before the TARGETS section. This is an example of how you can set your diversity in your LOCAL_CFLAGS and library name:

```
#-----------------------------------------------------------------
# Find string '_flo_' or '_int_' from the diversity environment
# variable _TMDIVERSITY
#-----------------------------------------------------------------
ifeq ($(findstring _flo_, $(_TMDIVERSITY)),_flo_)
LOCAL_CFLAGS = -DTMCOMP2FLO
LIB_SUFFIX = _flo
endif
ifeq ($(findstring _int_, $(_TMDIVERSITY)),_int_)
LOCAL_CFLAGS = -DTMCOMP2INT
LIB_SUFFIX = _int
endif
```

The LIB_SUFFIX is added to the library name. A LIB_SUFFIX has to start with an underscore (_) and has to have underscores between its diversities. Despite this, some diversities cannot be present. The developer determines the correspondence between the _TMDIVERSITY variable and the file name settings defined in LIB_SUFFIX. For example, you can decide that if you have _sp_ (single processor mode), LIB_SUFFIX will be empty.

Note: These settings are additions to what is set in the diversity.mk file (see Section 3.15.4, Component diversity.mk on page 99). The preferred style is to put certain diversity specifications in diversity.mk.

3.15.1.9 Local C, C++, LD, and TMTGT Flags

The list of flags below can be set in the component makefile to customize the compiler settings. In SDE2, TMTGT flags are located after option files.

```
LOCAL_CFLAGS =
LOCAL_CXXFLAGS =
LOCAL_LDFLAGS =
LOCAL_INCLUDES =
_TMTGTCOPTS =
_TMTGTCXXOPTS =
_TMTGTAOPTS =
_TMTGTINCLUDES =
```

<LOCAL_CFLAGS><_TMTGTCOPTS> （<LOCAL_CXXFLAGS><LOCAL_CXXFLAGS>）are specific flags added in this order to the compilation line when you compile for C (C++). LOCAL_LDFLAGS are added to the compilation of the executable. _TMTGTAOPTS is added to the assembler compilation line. For _TMTGTINCLUDES and LOCAL_INCLUDES read Section 3.4.5, Include directories on page 34.

Here you can put all your specific settings, as they are described in Table 3-10, Table 3-11, Table 3-12, Table 3-13, Table 3-14 and Table 3-15.

You can also include your Java settings. Java settings are described in Appendix A. The only mandatory Java setting is the location of the JDK. For example,

```
JAVATOP=c:/jdk1.3
```

3.15.1.10 Auto-documentation

You need this optional part if you want to have better documentation output generated by Doxygen in HTML and LaTeX format. You can write the following:

```
DOC_COMPNAME = tmComp1
DOC_SECTIONNUMBER = 01
```

3.15.1.11 Target(s)

This part is mandatory. You have to specify your goals in all. You have to have a configuration goal and at least one of the goals java, lib or target. Examples are:

```
all: configuration lib
all: configuration java lib
```

If your target is an executable, this part will look like:

```
TARGET = test
all: configuration target
```

You can have other goals as well. As an example if you have a diversity, you must declare the `diversity` goal like this:

```
all: diversity configuration lib
```

If you have a diversity goal, you must implement it in the file **diversity.mk** (preferably) or here. In case you implement the goal in **diversity.mk**, do not forget to put the following line in the component makefile:

```
include diversity.mk
```

This shows an example of a local implementation:

```
#-----------------------------------------------------------------
# Two kinds of diversities are supported
#-----------------------------------------------------------------
diversity:
ifeq ($(LIB_SUFFIX), )
        @$(ECHO) "_TMDIVERSITY must contain _flo_ or _int_"
        @$(ECHO)
endif
The other approach is to use recursive make. Example:
all: configuration FORCE
                @$(ECHO) making multi processor version
                @$(MAKE) "COMP7_LIB=lib_mp" "C_SOURCES=$(C_SOURCES_mp)" \
                        "LOCAL_CFLAGS=$(LOCAL_CFLAGS_mp)" lib
                @$(ECHO) making single processor version
                @$(MAKE) "COMP7_LIB=lib" "C_SOURCES=$(C_SOURCES_sp)" \
                        "LOCAL_CFLAGS=$(LOCAL_CFLAGS_sp)" lib
```

3.15.1.12 Makejava, makelib or maketarget

This part is mandatory. It consists at least of one of the **makejava** include, **makelib** include or **maketarget$(_TMBSL)** include. The includes **makelib** and **maketarget$(_TMBSL)** are not designed to be used together. Examples of the usage are:

```
#*********************************************************************
# Do not change the following include
#*********************************************************************
include $(DIR_SDE)/makejava.mk
and / or
ifneq ($(DIR_CONFIG),_)
include $(DIR_SDE)/$(DIR_CONFIG)/makelib.mk
endif
and / or
ifneq ($(DIR_CONFIG),_)
include $(DIR_SDE)/$(DIR_CONFIG)/maketarget$(_TMBSL).mk
endif
```

3.15.1.13 Different file specific compiler settings

This section is optional. You can specify different file compile settings per file, for example,

$(DIR_INTERM)/src/tmComp4int1.$(_SDE_O) : TARGET_CFLAGS+=-DTMCOMP4INT1

$(DIR_INTERM)/src/tmComp4int2.$(_SDE_O) : TARGET_CFLAGS+=-DTMCOMP4INT2

3.15.2 The += assignment

The += assignment is introduced for some makefile variables instead of the = assignment. The main difference is that you can set the initial value of your environment or makefile variable in your makefile and this value will be taken into account by SDE2. For example, if you need to add some extra object files to the linker, you can do this by setting **_SDE_OBJECTS** to the object file's location and name. If you do this for a variable initialized by =, the SDE2 will override your values and no action will be taken.

The following variables are initially assigned with +=:

- **_SDE_DEPENDENCIES** – Dependency files with.**d** extension.
- **_SDE_DLL_OPTIONS** – Here you can add additional DLL compile options. The DLL options are placed in the beginning of the line.
- **_SDE_LIB_PATHS** – Here you can add additional library paths.
- **_SDE_OBJECTS** – Here you can add additional object files.
- **_SDE_OPTIONFILES** – See the description in *Table 3-12 Variables for any makefile*.
- **PSOSOBJ** – pSOS object files needed for **mips_psos** executables.
- **VPATH** – Here you can add extra paths for your sources.
- **_SDE_DEP_MAKEFILES** – List of SDE2 used makefiles and component directories.
- **_SDE_WARNINGS** – List of SDE2 warnings, if any.
- **_SDE_REQUIRED_PATH_DLLS** – List of directories for SDE2 (DLL) binary release.
- **_SDE_REQUIRED_PATH_LIBS** – List of directories for SDE2 (library) binary release.
- **_SDE_IDLVPATH** – List of paths for IDL files.
- **_SDE_IDLFLAGS** – Flags for IDL compilation.
- **_SDE_CE_DLLS** – Required DLLs for WinCE recompilation.
- **_SDE_PROC_DEFINES** – Specific options for CPU type for the respective configurations.

3.15.3 Tables of all environmental and makefile variables

These tables do not include the Java environmental and makefile variables. The Java-related variables can be found in *Appendix A*.

Note: Do not change these values in your makefiles (exceptions are **_TMTGTAOPTS**, **_TMTGTCOPTS**, **_TMTGTCXXOPTS**, and **_TMTGTINCLUDES**).

Note: Changing the other values may lead to unpredictable behavior, especially if you use build scripts.

All variables in the Table 3-10 and Table 3-11 are environment variables.

This table lists the external target configuration variables and their descriptions.

Table 3-10: Variables for external target configuration

Variable	Purpose
_ECHOMAKELINES	Print echo messages if it is set
_SDE_VERSION	Version of SDE2
_TMBSL	SDE2 Board support library. See Table 3-5.
_TMCLOCKFREQ	TriMedia clock frequency; if undefined, we use 100MHz. TriMedia pSOS image build specific.
_TMDIVERSITY	SDE2 diversity. See Table 3-5.
_TMECHO	Print echo messages if it is set
_TMENDADDR	End address of memory image. TriMedia pSOS image build specific.
_TMLINKTYPE	SDE2 link type, currently used only for tm_psos configuration. See Table 3-5.
_TMMMIOBASE	Base address of MMIO; if undefined, we use **0x1be00000**. TriMedia pSOS image build specific.
_TMNODEPENDENCIES	Skip dependency generation, if set. See Table 3-5.
_TMNESTEDINCLUDE	Recursively close REQUIRES section, if set. See Table 3-5.
_TMPROJECT	This environment variable in used under cadenv environment in order specify the location of user specific settings (eg. $(UNAME).mk, prjlist.txt, buildlist.txt, configurations.txt etc) and also maketarget_$(TMBSL).mk files
_TMROOT	SDE2 template root. See Table 3-5.
_TMSITE	SDE2 site name. See Table 3-5.
_TMSTARTADDR	Start address of memory image. TriMedia pSOS image build specific.
_TMTOOLCHAIN	Tool chain for arm configuration. Possible values: **undefined** (default) or **ads** (arm specific).
_TMTGTAOPTS	Added at the end of the compilation line for assembler sources
_TMTGTBUILDROOT	SDE2 release root. See Table 3-5.
_TMTGTCOPTS	Added at the end of the compilation line for C sources
_TMTGTCOPYLIB	Option to generate binary release for libraries/DLLs. See Table 3-5.
_TMTGTCOPYOBJ	List of object files to be copied in the binary release. See Table 3-5.
_TMTGTCPP	Option to generate preprocessing information (for code debug).
_TMTGTCPUCLASS	CPU class. See Table 3-1. Should not contain capitals.
_TMTGTCPUTYPE	CPU type. See Table 3-1
_TMTGTCXXOPTS	Added at the end of the compilation line for C++ sources
_TMTGTENDIAN	Big- or little-endian. See Table 3-5.
_TMTGTINCLUDES	Added at the end of include directories path with **–I** prefix for each include directory.
_TMTGTOS	OS type. See Table 3-1
_TMTGTOSCLASS	OS class. See Table 3-1. Should not contain capitals.
_TMTGTREL	Release mode. See Table 3-5.
_TMDOC	Comma separated values for generating auto documentation. (pdf, html, tex and rtf)

Table 3-10: Variables for external target configuration

Variable	Purpose
_TMTGTWARNINGS	Warning mode. Values 0, 1, 2, 3 (default). Value 0 means no warnings (if possible), value 3 (almost) all possible warnings. Values 1 and 2 are intermediate. See *Section 3.4.7, Warning levels* on page 36.
_TM_C_ASM_CORREL	Used to switched on **armads_nullos**–specific feature that allows the generation of some extra support files. These files are used when the program is run on an IC design simulator as the target. Values: defined or undefined.
PATH	Search path. See Table 3-5.
TMP	Temporary directory. **Note:** This environment variable has to be defined for Windows 2000/XP.
UNAME	Name of the makefile with settings of the executables. See Table 3-5.

Table 3-11 lists the external environment configuration variables and their descriptions.

Table 3-11: Variables for external environment configuration

Variable	External environment
_FLATRELEASEDIR	CE-specific. Location of the **WinCE300/release** directory.
_WINCE_HPC_LIB_MIPS	Location of the **WCE200/MSHPC~1/lib/mips** directory.
_WINCEROOT	Location of the **WINCE** (**WinCE300**) directory.
DFP	MIPS-pSOS–specific setting for executables. Possible values: **S** or **H**.
DIABLIB	MIPS-pSOS–specific. Location of the **DIAB** libraries.
GCC_BASE	Path to GCC installation (for arm/mips/x86gnu_linux configurations)
GCC_PREFIX	The Cross compiler prefix (for arm/mips/x86gnu_linux configurations)
GCC_VERSION	GCC version being used (for arm/mips/x86gnu_linux configurations)
GHS_HOME	MIPS-INTEGRITY specific. Location of integrity OS installation.
ISIMIP	MIPS-pSOS–specific. Location of the isimip installation.
LM_LICENSE_FILE	MIPS-pSOS–specific. License file.
PSS_BSP	MIPS-pSOS–specific. Location of the **bsps/p4032** directory.
PSS_ROOT	MIPS-pSOS–specific. Location of the **pssmip.250** directory.
TCS	TriMedia specific. Location of the TriMedia installation.
TMPDIR	TriMedia-specific. The compiler creates temporary files in the current directory. If you do not have write privileges in this directory, the compiler will generate an error.
VCC	NT-specific. Location of the Visual Studio Installation.
WIND_BASE	Location of the **Tornado** directory.
WIND_HOST_TYPE	**Tornado**-specific. Host type.

All variables in the Table 3-12, Table 3-13, and Table 3-14 are makefile variables.

Table 3-12: Variables for any makefile

Variable	Function
_SDE_AOPTFILE_DEPENDS	If it is set, the file **$(DIR_INTERM)/a.opt** depends on its value. By default it depends on the makefile.
_SDE_BTM	If set to 1, different behavior for building executables is present. For WinCE platforms, the standard libraries **coredll.lib** and **corelibc.lib** are not included by default. For **mips_psos**, a ***.dld** file is not built and you must specify the target in your **maketarget$(_TMBSL).mk** specific makefile.
_SDE_COPTFILE_DEPENDS	If it is set, the file **$(DIR_INTERM)/c.opt** depends on its value. By default it depends on the makefile.
_SDE_CXXOPTFILE_DEPENDS	If it is set, the file **$(DIR_INTERM)/cxx.opt** depends on its value. By default it depends on the makefile.
_SDE_EXTRA_CFLAGS	Project-wide compiler-specific -D options for C files.
_SDE_EXTRA_CXXFLAGS	Project-wide compiler-specific -D options for C++ files.
_SDE_LOPTFILE_DEPENDS	If it is set, the file **$(DIR_INTERM)/l.opt** depends on it, otherwise it depends on the makefile.
_SDE_OPTIONFILES	List of the option files (**$(DIR_INTERM)/<c|cxx|a|l>.opt**) to be generated. The **make** utility generates only the required files; if an extra file is needed, it must be put in this makefile variable.
C_SOURCES	List of the names of all C source files if any. They must be located in the **src** subdirectory so the **src/** prefix must be present in the names.
CFG_SOURCES	List of the names of all configuration source files if any. They must be located in **cfg** subdirectory.
CXX_SOURCES	List of the names of all C++ source files if any. They must be located in **src** subdirectory.
DIR_CONFIG	Name of the configuration directory in **/sde**. The specific configuration files (**common.mk**, **makelib.mk**, **maketarget.mk**) are located there. Equals to **$(_TMTGTCPUCLASS)$(_TMTOOLCHAIN)_$(_TMTGTOSCLASS)**
DIR_INCLUDE	List of the include directories without a **–I** prefix. This list is put in the beginning of the whole include list.
DIR_LOCAL	Local directory. Usually set to **comps/tm<Component Name>**
EXTERNAL_DLLS	List of used external DLLs
EXTERNAL_LIBS	List of used external libraries
LIB_SUFFIX	Additional suffix to the library file for diversity purposes.
LIBS	List of used libraries and DLLs within SDE2; If you specify a component in this list, it means you are using a function from its public interface. The library of the used component is always listed? after your component's library in the linking line of the executable.
LOCAL_CFLAGS	Local flags added to the compilation line for C files.
LOCAL_CXXFLAGS	Local flags added to the compilation line for C++ files.
LOCAL_INCLUDES	Include directories set before the platform-specific directories. They can overwrite platform specific items.

Table 3-12: Variables for any makefile <Helv9R>(Cont'd.)

Variable	Function
REL_SUFFIX	Derived from _TMTGTREL. It can be _g, _a, or empty. Changing this variable will force a build in different compilation mode.
REL_COMPS_SUFFIX	Derived from _TMTGTREL. It can be _g, _a, or empty. Do not change this variable.
REQUIRES [1]	List of the names of the required components if any.
S_SOURCES	List of the names of all assembler source files (if any) located in or below the directory `src`.
TARGET_AFLAGS	Added in the compilation line for assembler sources
TARGET_CFLAGS	Added in the compilation line for C sources
TARGET_CXXFLAGS	Added in the compilation line for C++ sources

[1] It can be set to: REQUIRES = $(REQUIRED_INTERFACES) $(PROVIDED_INTERFACES)

Table 3-13 lists the library makefile variables and their descriptions.

Table 3-13: Variables for library makefiles

Variable	Description
_SDE_DLL_OPTIONS	Option for the DLLs generation. It is defined with +=, so you can add some options before the default ones.
DOC_COMPNAME	Used by Doxygen for automatic generation of the documentation.
DOC_SECTIONNUMBER	Used by Doxygen for automatic generation of the documentation.
EXPORTS	See section
LOCAL_DLLFLAGS	This flag is put in the end of _SDE_DLL_OPTIONS.

Table 3-14 lists the executable makefile variables and their descriptions.

Table 3-14: Variables for executable makefiles

Variable	Description
_SDE_SYSLIBS	mips_psos–specific default local target libraries. You can overwrite them.
LOCAL_LDFLAGS	Added at the end of the linker line.
LOCAL_SYSLIBS	mips_psos–specific custom target libraries. They are put before the default libraries.
PSS_CONFIG	mips_psos–specific. If it is defined PSS_CONFIG is not set to the default value.
TARGET	Target name

Table 3-15 contains a list of makefile variables set in the SDE2. You can use them in your makefiles. You can override them, but we do not support problems arising from overrides.

Table 3-15: Makefile variables used in SDE2

Variable	Description
_SDE_ALL_OBJECTS	All present objects in the release tree, used for the executable build. This variable is set and used only for x86_vxworks to resolve a specific issue.
_SDE_AOPTS	User assembler options.
_SDE_AOPTS_FILEX, X=1..4	SDE2 assembler options.

Table 3-15: Makefile variables used in SDE2 <Helv9R>(Cont'd.)

Variable	Description
_SDE_ARSUFFIX	Equal to REL_SUFFIX. Present in the target file name.
_SDE_ASM_LIST	Used to generate ASM disassembly of object files. armads_nullos–specific.
_SDE_BIN_REL_OBJ	List of object files, retrieved from a binary release.
_SDE_C_LIST	Used to generate C language disassembly of object files. armads_nullos–specific.
_SDE_CE_DLLS	Required DLLs for WinCE compilation
_SDE_COMMCTRL	CE–specific. Location of COMMCTRL library
_SDE_COPTS	User C options.
_SDE_COPTS_FILEX, X=1..4	SDE2 C options.
_SDE_COREDLL	CE–specific. Location of COREDLL library
_SDE_CPP_FILES	If _TMTGTCPP is set, it contains the list of files to store the preprocessed information.
_SDE_CPUTYPES_8051	Contains list of CPU types supported for 8051 CPU class
_SDE_CPUTYPES_arm	Contains list of CPU types supported for arm CPU class
_SDE_CPUTYPES_hp	Contains list of CPU types supported for hp CPU class
_SDE_CPUTYPES_mips	Contains list of CPU types supported for mips CPU class
_SDE_CPUTYPES_real	Contains list of CPU types supported for real CPU class
_SDE_CPUTYPES_tm	Contains list of CPU types supported for tm CPU class
_SDE_CPUTYPES_x86	Contains list of CPU types supported for x86 CPU class
_SDE_CXXOPTS	User C++ options.
_SDE_CXXOPTS_FILEX, X=1..4	SDE2 C++ options.
_SDE_DEBUG_OPTIONS	Contains extra debug options.
_SDE_DEP_MAKEFILES	List of SDE2 used makefiles and component directories. Used to force the update, if one of those files is changed.
_SDE_DEPENDENCIES	List of all dependency files if _TMNODEPENDENCIES is not set.
_SDE_DIR_BIN	Set to <_SDE_DIR_BUILD>/<_SDE_DIR_EXE_EXTPATH>
_SDE_DIR_BIN_EXTPATH	Set to bin/<DIR_CONFIG>_<_TMLINKTYPE>_<_TMTGTENDIAN>_ <_TMTGTCPUTYPE><LIB_SUFFIX><_SDE_ARSUFFIX>_ <_SDE_TCSHOST><_TMBSL>. _<_SDE_TCSHOST> present only in case CPU type is tm.
_SDE_DIR_CONFIGURATION	Set to <_TMTGTBUILDROOT>/<DIR_LOCAL>/cfg
_SDE_DIR_LIB	Set to <_SDE_DIR_BUILD_ROOT>/<_SDE_DIR_LIB_EXT>
_SDE_DIR_LIB_EXT	Set to lib/<DIR_CONFIG>_<_TMTGTENDIAN>_<_TMTGTCPUTYPE>
_SDE_DIR_LIB_LOCAL	Set to <_SDE_DIR_BUILD>/<_SDE_DIR_LIB_EXTPATH>
_SDE_DIR_REL_TO_LOCAL_ROOT	Relative location of the local root.
_SDE_DIR_REL_TO_ROOT	Undocumented.
_SDE_DIR_SED_SCRIPT	By default set to 's!\([^/]*\)\([/]*\)!..\2!g'. Can be set to another value in case sed misbehaves on your installation [1].
_SDE_DIRLIB_SUFFIX	Dir lib statement for the path of the libraries.

Table 3-15: Makefile variables used in SDE2 <Helv9R>(Cont'd.)

Variable	Description
_SDE_DIVERSITY	The name of the subdirectory below `lib` where DLLs are placed. Is set to a concatenation of the values of _TMDIVERSITY (minus trailing underscore) and REL_SUFFIX. In case both variables are undefined, is set to the current directory (.).
_SDE_DLL_PREFIX	Equals lib for tm_psos, otherwise undefined.
_SDE_DLLTARGETNAME	SDE2 DLL target name.
_SDE_ERROR	If it is not empty, prints an error message and stops.
_SDE_EXT_LIBS	List of the external libraries with their full path.
_SDE_FLAVOR_OPTIONS	Contains settings such as -DTMFL_REL=TMFL_REL_RETAIL
_SDE_GCC_AOPTIONS	Filtered –D, –U and –I assembler options.
_SDE_GCC_COPTIONS	Filtered –D, –U and –I C options.
_SDE_GCC_CXXOPTIONS	Filtered –D, –U and –I C++ options.
_SDE_IDLFLAGS	Flags for IDL compilation.
_SDE_IDLVPATH	List of directories for IDL files.
_SDE_IMPORT_DLLS	Contains list of all DLLs without their diversities, recursively searched in the required libraries list. If a DLL is found, it is put in this variable and there is no recursive search below this DLL.
_SDE_IMPORT_LIBS	Contains the list of all recursively required libraries without their diversities.
_SDE_IMPORT_REQUIRES	Contains the list of all recursively required interfaces without their diversities.
_SDE_INCLUDES	List of the include paths with –I prefix.
_SDE_INTERFACES [2]	Has to be set if it is needed.
_SDE_INTFS_IDL_DIRINC	Header files release directory (generated by IDL)
_SDE_INTFS_IDL_SOURCES	IDL source files
_SDE_INTFS_IDL_TARGETS	Header files generated from IDL source files and IDL tool.
_SDE_LBOPTS	User linker options (libraries).
_SDE_LDOPTS	User linker options (executable).
_SDE_LIB_CONFIGURATION	Set to $(DIR_CONFIG)_$(_TMGTENDIAN)_$(_TMTGTCPUTYPE).
_SDE_LIB_PATHS	SDE2 library generic path.
_SDE_LIBC / _sde_libc	CE–specific. Location of LIBC library
_SDE_LIBRARIES	List of the libraries with their suffixes and prefixes.
_SDE_LIBS_DIVERSITY	All recursively required libraries with their suffixes; the DLLs are not included.
_SDE_LIBS_NOSFX	Set to all required libraries without the part starting with an underscore.
_SDE_LINKSTYLE	Microsoft, nullos, linux or TriMedia compilation/link style.
_SDE_LINKTYPE	Set the linker options based on the linking type
_SDE_LIST_DIR	Intermediate location were some files are processed. armads_nullos–specific.

Table 3-15: Makefile variables used in SDE2 <Helv9R>(Cont'd.)

Variable	Description
_SDE_MI_FLAGS	Memory image build compilation flags. TriMedia memory image build specific.
_SDE_NCSCOPTS	NC-SC compiler options
_SDE_NCSC_FLAGS	NC-SC compiler flags
_SDE_NCSC_GCC	NC-SC gcc path
_SDE_NCSC_GXX	NC-SC g++ path
_SDE_O	Object extension name, usually **o** or **obj**.
_SDE_OSTYPES_ce	Contains list of OS types supported for **ce** OS class
_SDE_OSTYPES_cexec	Contains list of OS types supported for **cexec** OS class
_SDE_OSTYPES_ecos	Contains list of OS types supported for **ecos** OS class
_SDE_OSTYPES_integrity	Contains list of OS types supported for **integrity** OS class
_SDE_OSTYPES_linux	Contains list of OS types supported for **linux** OS class
_SDE_OSTYPES_mtos	Contains list of OS types supported for **mtos** OS class
_SDE_OSTYPES_nt	Contains list of OS types supported for **nt** OS class
_SDE_OSTYPES_nucleus	Contains list of OS types supported for **nucleus** OS class
_SDE_OSTYPES_nullos	Contains list of OS types supported for **nullos** OS class
_SDE_OSTYPES_oscan	Contains list of OS types supported for **oscan** OS class
_SDE_OSTYPES_psos	Contains list of OS types supported for **psos** OS class
_SDE_OSTYPES_ucos	Contains list of OS types supported for **ucos** OS class
_SDE_OSTYPES_vxworks	Contains list of OS types supported for **vxworks** OS class
_SDE_PROC_DEFINES	Specific options for CPU type for respective configurations
_SDE_PROVIDED_INTERFACES	SDE–provided interfaces.
_SDE_RECURSE_STATIC_LIBS	All recursively required components (libraries + dlls).
_SDE_REQUIRED_INTERFACES	List of required interfaces.
_SDE_REQUIRED_PATH_DLLS	Set to a list of existing directories containing the binary release of DLLs.
_SDE_REQUIRED_PATH_LIBS	Set to a list of existing directories containing the binary release of libraries.
_SDE_RGD_H_FILE	Name of the RGD header file.
_SDE_SUPPORTED_CPUTYPES	Contatins list of all SDE2 supported CPU types
_SDE_SUPPORTED_OSTYPES	Contatins list of all SDE2 supported OS types
_SDE_SUPPORTED_CPU_CLASSES	Contatins list of all SDE2 supported CPU classes
_SDE_SUPPORTED_OS_CLASSES	Contatins list of all SDE2 supported OS classes
_SDE_TARGET_SUFFIX	Extension of the executable/memory image build. TriMedia memory image build specific.
_SDE_TCSCPUTYPE	TriMedia-specific. Type of the CPU.
_SDE_TCSHOST [3]	TriMedia-specific. Host name. Equals to **_TMTCSHOST** if platform-specific **common.mk**, **makelib.mk** or **maketarget.mk** is included.
_SDE_THISDLL	Full DLL name.
_SDE_THISLIB	Full library name.

Table 3-15: Makefile variables used in SDE2 <Helv9R>(Cont'd.)

Variable	Description
_SDE_TMTGTBUILDROOT	The location of libraries and temporary files. It is set in **environment.mk** to the value of **_TMTGTBUILDROOT** (if defined) or to the value of **_TMROOT** (otherwise).
_SDE_VXWORKS_SUBDIRS	**x86_vxworks**–specific. List of component subdirectories, where the SDE2 has to look for object files to resolve C++ issue.
_SDE_WARNING_LEVEL	Compilation warning options.
_SDE_WARNINGS	SDE2 warnings.
AS	Assembler compiler.
CC	C compiler.
CXX	C++ compiler.
DIR_INTERM_EXE [4]	Set to <_SDE_DIR_BUILD>/tmp/<DIR_CONFIG>_<_TMTGTENDIAN>_<_TMTGTCPUTYPE><LIB_SUFFIX><_SDE_ARSUFFIX>_<_SDE_TCSHOST><_TMBSL>. The "_<_SDE_TCSHOST>" part is present if the processor is "**tm**".
DIR_INTERM_LIB	Set to <_SDE_DIR_BUILD>/tmp/<DIR_CONFIG>_<_TMTGTENDIAN>_<_TMTGTCPUTYPE><LIB_SUFFIX><_SDE_ARSUFFIX>
LB	Linker libraries.
LD	Linker executables.
PSOS_OBJS	Tm-specific. List of PSOS objects.
PSOSOBJ	**mips_psos**–specific. List of pSOS objects. The list is defined with += so you can add your MIPS objects too.

[1] A version of HP-UX 10.20 has been encountered with aberrant sed behavior.

[2] Example:
_SDE_INTERFACES= \
$(_SDE_REQUIRED_INTERFACES)
Note: The obsolete variable _SDE_INTERFACES has been replaced with _SDE_REQUIRED_INTERFACES.

[3] *Note:* _SDE_TCSHOST is initialized in **environment.mk** for tm platform to _TMTCSHOST. Later, for libraries, it is reinitialized in **tm_psos/common.mk** to **nohost**. For executables, it is the same. This does not cause any problems because _SDE_TCSHOST is used to set the paths only for executables.

[4] Be cautious using the recursive variables **DIR_INTERM_EXE** and **DIR_INTERM_LIB**. These variables refer to **LIB_SUFFIX** that may not be set and (for tm executables) **_SDE_TCSHOST**. So if you use them with an **ifeq** construction, set **LIB_SUFFIX** (if you have a different suffix) correctly before use.

Table 3-16:

Variable	Description
_<CompName>_DIR	Location of the component, including its name
_<CompName>_DIVERSITY	Component required diversities determine in run-time. This variable has to be located in diversity.mk
_<CompName>_LOADED	Trimedia specific. Component DLL loaded mode, default is "immediate".
_<CompName>_SUFFIX	Component required compilation mode. This variable has to be located in `diversity.mk`, if there is no default compilation mode.

3.15.4 Component diversity.mk

Each component can have a file **diversity.mk**. It is optional, but very important. The basic idea is to take some diversity definitions out of the makefile in order to make them accessible to other makefiles. The file is located in the same directory as the component **makefile** and may be considered part of it (meaning that if a component is delivered with the **makefile**, the **diversity.mk** file must also be delivered).

Note: The files **makefile** and **diversity.mk** are both part of a binary release.

The following can be done in the **diversity.mk** file in a component:

- Specify whether the component has to be required in a fixed compilation mode. You can do this by setting **_<CompName>_SUFFIX** to:
  - undefined or empty – default compilation mode
  - **_g** – debug compilation mode
  - **_a** – assert compilation mode
  - **_t** - trace compilation mode
  - **_r** – retail compilation mode

The **_<CompName>_SUFFIX** variable has to be set with **:=**.

- Extract the component diversity suffix from **_TMDIVERSITY**. This is done by setting **_<CompName>_DIVERSITY**. The **_<CompName>_DIVERSITY** variable has to be set with **:=**.
- Set **_<CompName>_LOADED** mode. This is a TriMedia-specific DLL loaded mode. It can be **immediate** or **deferred**.
- Set target **diversity**. Please note that this target should be set with double colon, i.e., **::**, because all components with diversity have this target and all targets should be checked.

Do not set any variables except these four in the file **diversity.mk**. This file is used directly by SDE2 makefiles and modifying other variables may cause unpredictable behavior. If you need intermediate variables, use unique names. .

The code below is an example of a **diversity.mk** file:

```
_Comp8_canon := $(findstring _mp_,$(_TMDIVERSITY))
_Comp8_canon := $(_Comp8_canon)$(findstring _sp_,$(_TMDIVERSITY))
_Comp8_canon := $(_Comp8_canon)$(findstring _td_,$(_TMDIVERSITY))
_tmComp8_DIVERSITY := $(patsubst %_,%,$(subst __,_,$(_tmComp8_canon)))
_tmComp8_SUFFIX := _r
diversity::
ifeq   (,$(filter _mp_ _sp_, $(findstring _mp_,    $(_TMDIVERSITY))$(findstring _sp_, $(_TMDIVERSITY))))
            @$(ECHO) "_TMDIVERSITY ($(_TMDIVERSITY)) must contain one of _mp_ or _sp_"
            @exit 1
endif
```

3.15.5 Reliable development with SDE2

As a rule, you set the definition of your environment variables in a batch file. You run this batch file and then you use/develop your components. However, in the process of development, you may want to change the value of one or more environment variable. The danger is, that this may not lead to the update of your dependencies, *.I, object and library files, i.e., your compilation process will be not correct.

Because checking whether or not the user has changed one of these environment variables may cost significant compilation time, this feature is not present in SDE2.

There are a few groups of environment variables and we will analyze each of these groups with respect to a reliable build process.

- Variables responsible for the configuration: _TMTGTCPUCLASS, _TMTGTOSCLASS, _TMTOOLCHAIN, _TMTGTCPUTYPE, _TMTGTOS, _TMTGTENDIAN, UNAME, _TMSITE, _TMBSL, _TMLINKTYPE, _TMPROJECT. It is not dangerous to change any of these variables, if you set a new proper value. You will just work with another configuration.
- Diversity variables: _TMTGTREL, _TMDIVERSITY. If you change one of these variables, you will get a warning/error message if SDE2 cannot handle some case properly. The usual warning message is that a *.I file is missing, because its location depends on those variables. It is not dangerous to change this variable if you take into account the warning messages.
- SDE2 process–specific variables: _TMTGTCOPYLIB, _TMTGTCOPYOBJ, _TMECHO, _TMTGTWARNINGS, _TMTGTBUILDROOT, _TMNODEPENDENCIES, _TMTGTCOPYOBJ, _TMTGTCPP, _ECHOMAKELINES, _SDE_VERSION, TMP. It is not dangerous to change these variables.
- SDE2 option variables: _TMTGTCOPTS, _TMTGTCXXOPTS, _TMTGTAOPTS, _TMTGTINCLUDES, LOCAL_INCLUDES, LOCAL_CFLAGS, LOCAL_LDFLAGS, ... It is dangerous to change these variables and if you do this, first clean your component directory.
- SDE2 root location in _TMROOT environment variable. It is unlikely and dangerous to change this variable and to work with multiple SDE2 copies.
- SDE2 tm_psos-specific memory build image environment variables: _TMMMIOBASE, _TMCLOCKFREQ, _TMSTARTADDR, _TMENDADDR. If you change one of those variables, it will not rebuild your memory image unless you first clean your image directory.

3.15.6 Tables of all examples included in the product

The SDE provides 21 example components and corresponding test applications. Each example shows some features of the SDE2. Table 3-17 describes per example its location, the features demonstrated, the corresponding test executables and the configuration classes the example has been tested for. If the Tested for column contains All, this means that the example is tested for all supported configurations.

Table 3-17: SDE2 examples

Example	Description	Tested for
comps/tmComp1	Basic library Example of source files with tags for autodocumentation with Doxygen and graphviz	All
comps/tmComp1/tst/Tst1	Basic test executable, also requires **tmComp2**. Test case for the **build.pl**.	All
comps/tmComp1/tst/Tst2	Basic test executable for **mips_psos**	**mips_psos** (PC Host)
comps/tmComp2	Components with user-defined diversity. Requires **tmComp1**. Test case for the **build.pl**.	All
comps/tmComp2/tst/Tst1	Executable using library **tmComp2** for a certain diversity	All
comps/tmComp3	Component using an abstract interface (**PROVIDED_BY** mechanism)	All
comps/tmComp3/tst/Tst1	Executable testing **tmComp3**	All
comps/tmRealFloat	Component implementing an abstract interface (**PROVIDED_BY** mechanism)	All
comps/tmComp4	Component with file-specific compile options	All
comps/tmComp4/tst/Tst1	Test application of **tmComp4**	All
comps/tmComp5	Component for SDE_in_SDE	All
comps/tmComp5/tst/Tst1	Executable for SDE_in_SDE	All
comps/tmComp6	Component with run-time diversity	All
comps/tmComp6/tst/Tst1	Test application that recompiled the run-time diversity of **tmComp6**	All
comps/tmComp6/tst/Tst2	Test application for object binary release	All
comps/tmComp7	Recursive make to save component diversity	**mips_psos** (PC Host) **tm_psos** (PC Host) **x86_nt** (PC Host) **mips_ce** (PC Host) **x86_ce** (PC Host)

Table 3-17: SDE2 examples <Helv9R>(Cont'd.)

Example	Description	Tested for
comps/tmComp7/tst/Tst1	Test application for **tmComp7**	**mips_psos** (PC Host) **tm_psos** (PC Host) **x86_nt** (PC Host) **mips_ce** (PC Host) **x86_ce** (PC Host)
comps/tmComp8	DLL component, always required in assert mode	**tm_psos** (PC Host) **x86_nt** (PC Host) **x86_ce** (PC Host) **mips_ce** (PC Host)
comps/tmComp8/tst/Tst1	Test application which uses **tmComp8** DLLs. Test example for warnings.	**tm_psos** (PC Host) **x86_nt** (PC Host) **x86_ce** (PC Host) **mips_ce** (PC Host)
comps/tmComp9	C++ test component; export classes in a DLL	All
comps/tmComp9/tst/Tst1	C++ test executable	**x86_nt** (PC Host)
intfs/ItmReal	Abstract interface of Real type. Used by **tmComp3**	**tm_psos** (PC Host) **x86_nt** (PC Host)
inc	Header files **tmCompId.h**, **tmAvFormats.h** and **tmtypes.h**; **mips_psos** directory	**tm_psos** (PC Host) **mips_psos** (PC Host)
comps/tmComp10	DLL-DLL test component, **LOADED** mechanism	Like **tmComp8**
comps/tmComp10/tst/Tst1	DLL-DLL test executable; **EXTERNAL_LIBS** test case	Like **tmComp8/tst/Tst1**
comps/tmComp11	MS Visual Studio integration example	**x86_nt** (PC Host)
comps/tmComp12	Java integration example	**x86_nt** (PC Host)
comps/tmComp13	Java JNI and NMI example.	**x86_nt** (PC Host)
comps/tmComp14	Complex DLL, see description below this table	All
comps/tmComp14/tst/Tst1	Executable for **tmComp14**, see description below this table	All
comps/tmComp15	Component required always in retail mode	All
comps/tmComp15/tst/Tst1	Executable for **tmComp15**	All
comps/tmComp16	Component implements **ItmDummy**. Generate DLL. Requires **tmComp8** and **tmComp1**.	All
comps/tmComp16/tst/Tst1	Override **tmComp1** rel.mode and **tmComp8** diversity.	All
comps/tmComp17	Component implements **ItmDummy**. Generate DLL. Requires **tmComp8** and **tmComp1**.	All

Table 3-17: SDE2 examples <Helv9R>(Cont'd.)

Example	Description	Tested for
comps/tmComp17/tst/Tst1	Replace in a link time **tmComp17** with **tmComp16**.	All
comps/phSimpleComp1	Test component for SystemC	x86ncsc_nullos
comps/phSimpleComp2	Test component for SystemC	x86ncsc_nullos
comps/phSimpleComp2/tst/tst1	Test Application for SystemC	x86ncsc_nullos

There are the following complex dependencies for **tmComp14/tst/Tst1**

tmComp14/tst/Tst1 requires tmComp14 (DLL)
tmComp14 (DLL) requires tmComp10 (DLL), tmComp15 (always retail), tmComp3
tmComp10 (DLL) requires tmComp8 (DLL, assert), tmComp2 (diversity: _flo), tmRealFloat
tmComp15 (retail) requires tmComp1, tmComp2 (diversity: _flo)
tmComp3 requires tmRealFloat

The building of **comps/tmComp14/tst/Tst1** can be executed with one command:

perl build_exe.pl comps/tmComp14/tst/Tst1

In this example the default compilation mode is used if explicitly specified (**tmComp8**, **tmComp15**). Compilation is done for a DLL if possible (**x86**, **ce**), for a static library otherwise.

3.16 User Configurable New CPU/OS Type

SDE2 provides to its users, an option to extend the support for new CPUs/OS types themselves, as and when the new CPU/OS types are available with out waiting for new SDE2 official release supporting the new CPU/OS types.

Inorder to support a new CPU/OS type the user needs to create a file named **new_cpu_os_type.mk** in the directory **$_TMPROJECT/$(DIR_CONFIG)/**. The contents should be as follows:

```
_SDE_CPUTYPES_$(_TMTGTCPUCLASS) = <newcpu1> <newcpu2> <newcpu3>
ifeq ($(_TMTGTCPUTYPE),<newcpu1>)
_SDE_PROC_DEFINES=<newcpu1 options>
endif
ifeq ($(_TMTGTCPUTYPE),<newcpu2>)
_SDE_PROC_DEFINES=<newcpu2 options>
endif
ifeq ($(_TMTGTCPUTYPE),<newcpu3>)
_SDE_PROC_DEFINES=<newcpu3 options>
endif
_SDE_CPUTYPES_$(_TMTGTOSCLASS) = <newos1> <newos2> <newos3>
ifeq ($(_TMTGTOSTYPE),<newos1>)
_SDE_PROC_DEFINES=<newos1 options>
endif
ifeq ($(_TMTGTOSTYPE),<newos2>)
_SDE_PROC_DEFINES=<newos2 options>
endif
ifeq ($(_TMTGTOSTYPE),<newos3>)
_SDE_PROC_DEFINES=<newos3 options>
endif
```

For Example:

```
D:\>set _TMPROJECT
_TMPROJECT=d:/project
D:\>dir d:\project\x86_nt\new_cpu_os_type.mk
Volume in drive D is ABCD
Volume Serial Number is ABCD-VXYZ

Directory of d:\project\x86_nt

08/09/2005  12:30 PM              51 new_cpu_os_type.mk
```

Contents:

```
D:\project\x86_nt>cat new_cpu_os_type.mk
_SDE_OSTYPES_nt += nt5
_SDE_CPUTYPES_x86 += i686
```

NOTE: This implementation doesnot include CPU/OS masking bits. In order to include the masking bits support user needs to send his latest **new_cpu_os_type.mk** to SDE2 team.

3.17 New third party toolset integration

SDE2 provides to its users, an option to extend the support to easliy integrate the new tools into SDE2 environment.

Users need to edit and update a file called project_include.mk located in the directory mentioned in **_TMPROJECT** environment variable (if set) or **_TMROOT/projecty/** directory. This file would either have the modifications/rules and commands for new tools or this file would include another makefile that has the rules and commands for the new toolset.

For Example:

```
D:\>set _TMPROJECT
_TMPROJECT=d:/project
D:\>dir d:\project
Volume in drive D is ABCD
Volume Serial Number is ABCD-VXYZ

Directory of d:\project

02/06/2006 12:30 PM              51 project_include.mk
02/06/2006 12:45 PM             141 docjet.mk
```

Contents:

```
D:\project>cat project_include.mk
-include docjet.mk

D:\project>cat docjet.mk
 docjet:
        @echo "Generating docjet documentation....."
        @d:/progra~1/talltree/docjet/Program/Generator -v d:/config.djt
```

Chapter 4
System C support in SDE2

User Manual Version 3.8 **Sep 29, 2006**

What is covered in this chapter?

This chapter provides information on the System C support provided by SDE2. This feature of SDE2 is mainly used to build software simulated hardware IPs

4.1 Introduction to System C

SystemC provides hardware-oriented constructs within the context of C++ as a class library implemented in standard C++. Its use spans design and verification from concept to implementation in hardware and software.

SystemC provides an interoperable modeling platform which enables the development and exchange of very fast system-level C++ models. It also provides a stable platform for development of system-level tools.

The Open SystemC Initiative (OSCI) is an independent not-for-profit organization composed of a broad range of companies, universities and individuals dedicated to supporting and advancing SystemC as an open source standard for system-level design.

The NC-SC™ Simulator is the industry's premier environment for transaction-level model development and verification. With performance 100x faster than equivalent RTL, transaction-level modeling is ideal for low-level embedded software development and architectural analysis. The NC-SC simulator supports the SystemC® Verification Library, making it ideal for creating transaction-level testbenches.

4.2 Supported Configurations

SDE2 supports the following system C configurations

- **x86ncsc_nullos** - NcSc System
- **x86osci_nullos** - OSCI System
- **hpncsc_nullos** - NcSc System

4.2.1 Cadence-NcSc System C support - HW Modeling (x86ncsc_nullos)

Inorder to support building of reusable System C components using Cadence NC-SC, for Hardware Modeling team a new configuration class x86ncsc_nullos has been included in SDE2. Linux being the host platform. The CPUCLASS and CPUTYPE for this configuration being x86 and i486 respectively, OSCLASS and OSTYPE being nullos. The toolchain is “ncsc”. The compiler used for dependency generation and compilation is ncsc.

4.2.2 OSCI System C support

Inorder to support building of reusable System C components using OSCI, for Hardware Modeling team a new configuration class x86osci_nullos has been included in SDE2. Linux being the host platform. The CPUCLASS and CPUTYPE for this configuration being x86 and i486 respectively and OSCLASS and OSTYPE being nullos. The toolchain is “osci”. The compiler used for dependency generation and compilation is gcc.

4.2.3 Cadence-NcSc System C support - NxBuilder Support

Inorder to support building of reusable System C components using Cadence NC-SC, for NxBuilder team a new configuration class hpncsc_nullos has been included in SDE2. HP-UX being the host platform. The CPUCLASS and CPUTYPE for this configuration being hp, OSCLASS and OSTYPE being nullos. The toolchain is “ncsc”. The compiler used for dependency generation and compilation is ncsc.

Chapter 5
Installation and Customization

User Manual Version 3.8 **Sep 29, 2006**

What is covered in this chapter?

This chapter provides information for installing and customizing SDE2 for your working environment, including:

- A list of required tools
- Setting permissions
- Tuning scripts and makefiles
- Using SDE2 with CMSynergy

5.1 Installation

The installation instructions are part of the Release Notes *[RELNOT] DVP SDE2 2.3 Release Notes,* (see Bibliography) and **SDE2_installation_notes_2.3.txt** file.

5.2 Customization

This section includes a list and description of the required tools, permissions, information about tuning initialization scripts, and instructions for creating a configuration class.

5.2.1 Required tools

Besides compilers (that may be project-specific), SDE2 requires some general tools for performing the **make** and the dependency checks. The following table lists the tools for the PC environment that are delivered with SDE2[7]. For the UNIX/Linux environment the same tools are available, but they are not supplied with SDE2, as they are UNIX/Linux platform dependent.

Table 5-1: PC environment tools required by and delivered with SDE2

Tool	Version	Comment
basename		
cat		
chmod		
cmp		
cp		

7. There are different deliveries of these tools for PC. We are using the standard Cygwin delivery, because it is shareware. Some deliveries, like MKS, perform better, but they are commercial products. So, if the you have a licence, you can use them. The standard Cygwin delivery consists of **cygwin1.dll** and tools. The current version of **cygwin1.dll** is 1.3.22. You can find more information in *Appendix F*.

Table 5-1: PC environment tools required by and delivered with SDE2 <Helv9R>(Cont'd.)

Tool	Version	Comment
diff		
dirname		
doxygen [1]	1.4.3 or later	Required for auto-documentation
echo		
gawk	3.0.4 or later	
gcc	2.95.3-5 [2] or later	Required for dependency checks
gmake	3.80	
ls		
mkdir		
pwd		
sed	GNU sed 3.02 or later	
sort		
tee		
test		
touch		
xargs		

[1] Doxygen is delivered without the **hhc.exe** tool. For more info about **hhc.exe**, read *Appendix D*.

[2] Note that version 2.95.2-5 assumes that **_WIN32** is always defined. For **tm_psos** and **mips_psos** it could cause problems. Therefore we do not support this version.

The following tools are the supporting tools the current release of SDE2 has been tested for. These tools are not delivered with SDE2. Each of these tools has set of subtools, such as compilers, linkers, etc. The versions of these subtools can be determined by the version of the tool. Note that SDE2 supports only one toolset. If you want to override some of the default settings (use another compiler, etc.), you can try, but we cannot guarantee that compiling will still work, and cannot support tools other than those suggested.

Table 5-2: Tools required by but not delivered with SDE2

Tool	Version	Comment
Adelante SDK	Latest version	Required for **realsat_nullos**
Arm-ads Toolset	1.1 Or Later	Required for **arm_cexec** and **armads_nullos**
Arm-elf Tool Set	2.96 Or Later	Required for **arm_nullos** only
Arm-Realview Toolset		Required for **armrvds_nullos**
Developer Studio	Version 6 or Later	Required for **x86_nt**
Ecos GNU toolchain		Required for **mipsgnu_ecos**
Graphviz	1.5 Or Later	Required for auto-documentation graphics
GreenHills Toolchain for INTEGRITY		Required for **mipsghs_integrity**
GreenHills Toolchain for osCAN		Required for **mipsghs_oscan**
GreenHills Toolchain		Required for **mipsghs_nullos**
Isi Diab Data Tool Set	4.3p5	Required for **mips_psos** only
Java 2 Runtime Environment	1.3	Required for java components

Table 5-2: Tools required by but not delivered with SDE2 <Helv9R>(Cont'd.)

Tool	Version	Comment
Nxpidl	2.0.0026 Or Later	Required for header files generation from idl files. this tool is delivered both for nt (in directory cygwin) and linux (in directory **cygwin/hp_nullos**).
Perl	5.005_03 Or Later	Required for build scripts
Rcc/toolchain For Rd24120	1.1.0 Or Later	Required for **real_nullos** and **real_mtos**
Tornado 2.1/2.2	V5.4 Or Later	Required for **mips_vxworks**, **arm_vxworks** and **x86_vxworks** only
Trimedia-psos Tool Set	V4.4 Or Later	Required for **tm_psos** only
Wince	Version 3.00	Required for **mips_wince**, **arm_wince**, **x86_wince**
Windows DDK		Required for **x86ddk_nt**

Table 5-3: Tools required by but not delivered with SDE2 for System C Components

Table 5-4:

Tool	Version	Comment
Cadence NC-SC on Linux Host		Required for **x86ncsc_nullos**
Cadence NC-SC on Linux Host		Required for **hpncsc_nullos**
GCC		Required for **x86osci_nullos**

5.2.2 Permissions

The following table describes what parts of (the **sde** directory of) the SDE2 environment may be changed by local sites.

Table 5-5: Overview of changeable parts of SDE2

Description of the modification	Location	Reason
Adding initialization scripts	**project/sites/<site>/*.bat**	Each site has unique attributes with respect to the location of tools, boards, etc.
Specifying location of tools	**project/sites/<site>/linux.mk** or **project/sites/<site>/cygwin.mk**	Each site has unique attributes with respect to the location of tools
Adding **maketarget<xxx>.mk**	**sde/<configuration-class>/maketarget<xxx>.mk**	Each site may have its own development board containing its own BSP and other standard libraries.
Adding / editing **project/prjlist.txt**	**project/prjlist.txt**	Describes each of the multiproject roots.

5.2.3 Tuning the SDE initialization script

For each platform there is an SDE initialization script (or batch file) that sets basic environment variables required by SDE2 and the compilers. In Table 3-5 more information regarding these variables can be found. Table 5-6 shows the predefined scripts and batch files that are distributed with SDE2. The scripts are in the directory **`project/sites/blrsdm`**.

Table 5-6: Standard initialization scripts of SDE2

Script	Used for host platform
arm_ce_debug_static_default.bat	PC
arm_vxworks_default.bat	PC
hp_nullos	UNIX
linux_nullos	Linux
mips_ce_debug_static.bat	PC
mips_psos_debug_static_eb_p4032.bat	PC
mips_vxworks_default.bat	PC
real_mtos_default.bat	PC
tm_psos_assert_static_el_tm32_winnt.bat	PC
tm_psos_debug_static_el_tm32_winnt.bat	PC
tm_psos_retail_static_el_tm32_winnt.bat	PC
x86_ce_debug_static.bat	PC
x86_nt_debug_static.bat	PC

Installing the SDE2 at your own site usually means taking one or a few of these scripts as a starting point and modifying them.

5.2.4 Tuning linux.mk

Unix hosts have different configurations for different sites. This means that tools like **gawk** may be in different locations per site. The **linux.mk** file contains the necessary information. Each site has to tune the **linux.mk** file to their local situation. For PC environments all tools are delivered by SDE2. The **cygwin.mk** file contains the locations of the relevant tools. This file is the same for all PC environments.

For Linux environments the following has to be done:

- Create an **`project/sites/<site>`** directory, where **`<site>`** is the identification of your site, for example **ehvblv** (Eindhoven Business Line Video).
- Copy the **`project/sites/blrsdm/linux.mk`** file to the **`project/sites/<site>`** directory. This file contains variables that refer to the location of the tools that are required by the SDE2, see *Table 5-7 Tools that are set by linux.mk and cygwin.mk*. For the PC environment, these tools are part of the SDE2 package. For UNIX environments, these tools are typically available at a certain location in your UNIX environment. For your UNIX environment you may have to change the location values of those variables.
- Initialize the environment with **UNAME=hpux** (see **hp_nullos** script) and **_TMSITE=<site>**.

The SDE2 will consult the `project/sites/<site>/linux.mk` file to find the correct tools.

5.2.5 Tuning cygwin.mk

For PC environments no extra actions are required, except that the `cygwin.mk` file of the directory `project/sites/blrsdm` has to be copied to your local directory `project/sites/<local site>`. After that, you can initialize SDE2 using one of the standard scripts, or a modified one, see *Section 5.2.3, Tuning the SDE initialization script* on page 110.

You can tune and use different utilities (and/or versions of these utilities) than those proposed by the SDE2 team. You can also add extra options to your utilities.

Table 5-7: Tools that are set by linux.mk and cygwin.mk

Variable	Description
_SDE_DOT_PATH	graphviz path executable. *Optional.*
CAT	catalog command
CD	change directory command
CHMOD	Change mode command
CMP	compare command
CP	copy command
DOXYGEN	Auto-documentation tool. *Optional.*
ECHO	echo command
GAWK	gawk utility
GCC	gnu c++ compiler
LS	List command
MAKE	make command
MKDIR	make directory command
RM	remove command
SED	sed command
SORT	Sort command
TEST	test command
TOUCH	touch command
XARGS	args command. *Optional.*

5.2.6 Tuning prjlist.txt

The file `prjlist.txt` describes the roots of all used multiproject trees. It is always located in the directory `<_TMROOT>/project`. The file may contain environment variables defined in a UNIX style, for example,
`$_TMTGTREQPRJ ${_TMCUSTOMPRJ}`

If the environment variables are used, you are responsible for setting the values of those variables. The standard absolute path locations can also be used for this file, for example,

c:/nexperia/infra c:/nexperia/kernel

If this file is not present, SDE2 automatically creates it with one default entry, the value of **$_TMROOT**.

If you edit /touch this file, **loc_list.*** files will be updated when you start SDE2 for the first time. This could lead to use of another component's locations.

This file can also contain comment lines, starting with #

5.2.7 Tuning sde directory

There are a few makefiles, whose features are rarely used in SDE2. These makefiles are located in the main **sde** directory (except **autodoc.mk**, which is located in the **sde/autodoc** directory) and they all are included from **sde/common.mk** with the **-include** option, which means that if they are not present, no warning message will appear. If you are sure that your project is not using any feature described below in one of its makefiles, you can remove the corresponding makefile and you will notice a small improvement in compilation time. However, if your project needs the makefile, you may see strange error messages. So, you are responsible for their removal and SDE2 does not warn you if some component uses one of those features.

These files are:

- **autodoc/autodoc.mk** – Contains rules to generate Doxygen autodocumentation output.
- **provided_by.mk** – Contains rules to handle **PROVIDED_BY** construction described in *Section 3.5.2, Complex interface diversity* on page 40.
- **qac.mk** – Please refer to *Appendix J* for more information QAC support in SDE2.
- **rgd.mk** – Contains undocumented rules for **rgd** tool.
- **idl.mk** – Contains rules for **idl** and header file generation
- **qmore.mk** – Contains rules for **qmore** invokation.
- **lint.mk** – Contains rules for **lint** invokation.

5.3 SDE2 on cadenv

Cadenv is NXP Semiconductors environment for CAD tools. SDE2 has been updated/modified inorder to support cadenv environment. This section helps in installing SDE2 on cadenv environment.

5.3.1 Installing SDE2 on cadenv

1. Log into the network as a software administrator.

2. Create the required directory structure under **/cadappl**:

% mkdir /cadappl/ictools/sde2

3. Set the working directory to this directory and transfer the tar file **sde2_2_3.tar.gz**

% cd /cadappl/ictools/sde2

% cp <path_to_sde2_tar_file> .

4. Expand the tar file:-

% gunzip -c sde2_2_3.tar.gz | tar -xvf -

!! This will create a directory **'sde_template'** containing the SDE2 installation. Move this directory to 2.3.

% mv sde_template 2.3

5. Ensure the protections of the newly unpacked files as necessary. Typically this would be rwxr_xr_x, :-

% /bin/chmod -R 755 2.3

6. Certain cadenv commands expect there to be a help file installed. If you have not created or installed one for this package, this needs to be completed before proceeding further with the installation. An updated help file is provided with this package, and should be installed as follows:-

% cadenv -H sde2 /cadappl/ictools/sde2/2.3/install/sde2.hlp

These files can be edited before or after installation for any site specific information. They are located in:- **/cadappl/cadenv/hlp/sde2**

7. Check the contents of the cadenv release file **'sde2.rel'** in the **'sde2/2.3/install'** directory. Create the cadenv release file as follows, or by using the method favoured by your site:-

% cadenv -C -r 2.3 sde2 /cadappl/ictools/sde2/2.3/install/sde2.rel

8. To make the package available to users it must be mapped onto a cadenv version. For example, it can be mapped to the "cur"rent version as follows:-

% cadenv -M -r 2.3 sde2 cur

Users can then install the "cur"rent SDE2 version with:-

% cadenv -c sde2

10. You can now archive or delete the tar file:

sde2_2_3.tar.gz

5.3.2 User specific customizations

SDE2 directory structure under cadenv is not writable by all users except for the system administrator. But each user needs to have write permission to the project directory under SDE2 installation directory to have his own settings. To work around this problems SDE2 has an environment variable **_TMPROJECT** which needs to be set when using SDE2 in cadenv environment. If this environment variable is set then SDE2 would look into the contents of this directory for local settings instead of **$(_TMROOT)/project** directory. The possible contents of the **$(_TMPROJECT)** directory are:

buildlist.txtlist of components to be built by build.pl script

configurations.txtconfiguration list supported by the project

prjlist.txtlocation of the project components

sites/site specific settings

$(DIR_CONFIG)/**user specific **maketarget_$(_TMBSL).mk files

5.3.3 SDE2 cadenv wrapper scripts

SDE2 delivers 4 wrapper scripts for cadenv. The following are the details of the wrapper scripts provided:

sdemake Wrapper script for gmake.

sdedocWrapper script for viewing SDE2 documentation

usage: sdedoc [start|manual|release] for Getting started manual SDE2

User manual and Release notes respectively

sdebuildWrapper script for build.pl perl script

sdebuild_exe Wrapper script for build_exe.pl perl script

5.3.4 Other software tools that are required in cadenv

Other tools that need to be cadenved are as follows:

gmake - 3.80

gcc -2.95.2 or later

acrobat -4.05 or later

perl-5.6.1 or later

5.4 SDE2 and CMSynergy

In using a component-based work environment, each component may have its own life cycle. Therefore it is logical that each component is mapped on a CMSynergy project.

These CMSynergy projects must be made into subprojects that are linked to an arbitrary main project, because this is the only way to maintain the directory structure proposed by this document.

The **sde** directory (containing all generic makefiles and scripts) possesses its own life cycle. This directory is also mapped as a CMSynergy project.

Below, is an example directory structure of a project **utv**, where each underlined directory is mapped on a CMSynergy project. Note that the first three directories are introduced by CMSynergy.

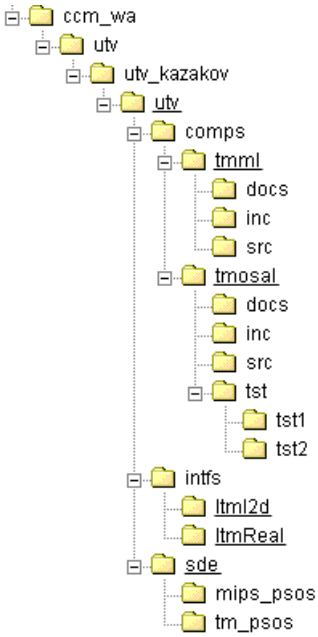


Figure 5-1: Directory structure of a project utv

Note that the CMSynergy delimiter (in this case -, it might be also =, etc.) cannot be a part of the names of directories or files. This is a CMSynergy requirement.

Appendix A
Java Building

User Manual Version 3.8 Sep 29, 2006

What is covered in this appendix?

This appendix describes a recommended standard way to organize components containing Java source code using SDE2, including:

- Java implementation and possible problems
- Java cross compilation and compilation class paths
- Implementation principles and limitations
- Information about user-configurable variables and internal SDE2 variables
- Searching the **javac** class and source files

A.1 Quick start

This is a Java makefile example:

```
#-----------------------------------------------------------------
# Makefile for java component
#-----------------------------------------------------------------

DIR_LOCAL   =   comps/...
include $(_TMROOT)/sde/environment.mk

#-----------------------------------------------------------------
# Source environment variables
#-----------------------------------------------------------------

# Default JAVA_SOURCEPATHS is src:
#JAVA_SOURCEPATHS = src/some_configuration src

JAVA_CLASSES = \
            com.nxp.sde.Test1 \
            com.nxp.sde.Test2

JAVA_JNICLASSES = \
            com.nxp.sde.Test1

JAVA_NMICLASSES = \
            com.nxp.sde.Test2

C_SOURCES = \
```

```
            src/com/nxp/sde/Test1.c \
            src/com/nxp/sde/Test2.c

# For jni.h:
DIR_INCLUDE += $(JAVATOP)/include
# for generated jni/nmi files:
DIR_INCLUDE += $(JAVA_INCDIR)

#-----------------------------------------------------------------
# Which other Java components does this target require to compile
# Append $(JAVA_REL_SUFFIX) if you want the same configuration as this
# component.
#-----------------------------------------------------------------
JAVA_REQUIRES = tmComp1$(JAVA_REL_SUFFIX)

#-----------------------------------------------------------------
# System class paths
#-----------------------------------------------------------------
# Required: set JAVA_BOOTCLASSPATHS and/or JAVA_EXTDIRS:
# Note: Should use to JVM class path used on embedded target:
# Possibly set outside this makefile.
#JAVA_BOOTCLASSPATHS =
JAVA_EXTDIRS =

#-----------------------------------------------------------------
# Directories/jars/zips where the 3rd party classes are stored
# These are added at the end of -classpath.
#-----------------------------------------------------------------
LOCAL_CLASSPATHS =

#-----------------------------------------------------------------
# local FLAGS
#-----------------------------------------------------------------
LOCAL_JAVACFLAGS =
LOCAL_JAVAHFLAGS =
LOCAL_JAVAJARFLAGS =
LOCAL_CFLAGS =
LOCAL_CXXFLAGS =

#-----------------------------------------------------------------
all: configuration java lib

#*****************************************************************
# Do not change the following include
#*****************************************************************
include $(DIR_SDE)/makejava.mk
ifneq ($(DIR_CONFIG),_)
include $(DIR_SDE)/$(DIR_CONFIG)/makelib.mk
endif
```

A.2 Java implementation

A.2.1 Relevant problem aspects

The following aspects are relevant for building Java code:

1. Which java source files need compiling.
2. Compatibility of output: JDK 1.1, 1.2 or 1.3.
3. Debug options.
4. Classpath for Java system libraries. Usually you use cross-compilation for embedded systems, and so you need to be careful which system class path you use (**javac** option **-bootclasspath** and **-extdirs**)
5. Classpath for Java application reference libraries.
6. Where to store results (classes directory, jni/nmi header files).
7. Native methods, and JNI and NMI header file generation.
8. Which Java compiler is used, and which javah/JavaJar program is used.
9. Name(s) of output JAR file
10. Command-line length problems

A.2.2 Java cross-compilation

Cross-compilation is usually used with embedded applications. For Java that means you need to provide the **-bootclasspath** and/or **-extdirs** options for system libraries. Ideally the embedded JVM is provided as an SDE2-compatible component.

To help ensure that users do not inadvertently use JDK system class libraries, the SDE2 requires the definition of **JAVA_BOOTCLASSPATHS** and/or **JAVA_EXTDIRS** variables.

A.2.3 Java compilation class path

Java files in a component may need Java classes/JAR files from other components. This is provided for as follows:

1. If the referenced Java classes are part of another SDE2 component, then just add the name of the component to the **JAVA_REQUIRES** variable (as for C-interface dependencies between components, with optional **$(JAVA_REL_SUFFIX)**).
2. For external class libraries, add directories, JAR and zip files to the **LOCAL_CLASSPATH** variable.
3. SDE2 will form the **-classpath** option value from:
   a. **LOCAL_CLASSPATHS0**
   b. **JAVA_CLASSESDIR** – this is where the component will generate its own class files
   c. **LOCAL_CLASSPATHS1**
   d. JAR file list for each of the **JAVA_REQUIRES**
   e. **LOCAL_CLASSPATHS**

A.2.4 Implementation principles

Certain principles govern the implementation guidelines:

1. Being able to redefine almost every option.
2. Having useful default values if possible.
3. Use of prefixes in make variables and internal targets.
4. Check for undefined or empty variable errors: generate error and stop.

A.2.5 Implementation limitations

There are some limitations and implementation choices:

1. A component can only make one JAR file. (Same restriction as for # object libraries.)
2. The name of the resultant JAR file is **<component_name>[<suffix>.jar**. The suffix **$(JAVA_REL_SUFFIX)** is **_g** for debug builds. (Component name is, until new SDE2 release: **tail** of **$(DIR_LOCAL)**).

A.2.6 Implementation approach

A problem exists with specifying which Java files to compile, and then in specifying dependencies for the resultant **.class** file. Given a Java source file name, the name of the class is derivable, but you cannot determine the fully qualified class name including the package name. For that you have to look into the Java source file.

The Sun **pJava** makefiles solve this by specifying the required fully qualified class names that need to be compiled. Using textual substitution you can derive names of class files and Java files. The Java file names are not necessarily in the right directory, therefore use the **VPATH** construct to allow make to find them.

The approach is thus:

1. The user's makefile defines **JAVA_CLASSES**.
2. SDE2 defines **JAVA_CLASSFILES** from **JAVA_CLASSES** by changing . to /, putting **$(JAVA_CLASSESDIR)** in front and appending **.class**.
3. The user's makefile adds the **java** target to the **all** target dependencies before a **lib** target.
4. The user specifies the required Java components in **JAVA_REQUIRES**. The search path in order of priorities is:
   - **lib/jar** release directory
   - **lib/jar** component directory (binary release support)
5. SDE2 defines Java dependent on **$(JAVA_CLASSFILES)** and has a rule for running **$(JAVAJAR)**.
6. SDE2 defines pattern rule **$(JAVA_CLASSESDIR)/%.class : %.java**, which if invoked will echo the Java file name to a temporary Java file list file.
7. The temporary Java file list file is passed to **$(JAVAC)**.

8. After compilation **$(JAVAJAR)** is run, archiving the complete **$(JAVA_CLASSESDIR)** directory.

Native methods need special provisions. This is catered to as follows:

1. Any class having native JNI methods needs to be added to the variable **JAVA_JNICLASSES**.
2. Any class having native NMI methods needs to be added to the variable **JAVA_NMICLASSES**.
3. The Java target will generate include files for these in the `inc/jni`[8] or `inc/nmi`[9] subdirectories of the component build tree. For example if you add **com.nxp.sde.Test1** to **JAVA_JNICLASSES**, the file **inc/jni/com_nxp_sde_Test1.h** will be generated.
4. To use the generated `jni/nmi` files, you have to add one or more paths to your **DIR_INCLUDE** variable. By default you should always include your **jni** files as **#include "jni/Java_.....h"**, and add **$(JAVA_INCDIR)** to **DIR_INCLUDE**. However, if you do things differently, then this depends on whether you have overridden the default values of **JAVA_JNIDIR** and/or **JAVA_NMIDIR**, and on whether you include **jni** or **nmi** files without the **jni/** or **nmi/** prefix. You should also ensure that the relevant **jni.h** (and **jni_md.h**) can be found; **jni.h** often sits in **`$(JAVATOP)/include`**. Which **jni_md.h** should be used, depends on your compiler.

A.2.7 User-configurable variables

All user-configurable variables are never redefined by the SDE2. Table A-1 lists the variables needed to build Java components.

Table A-1: Required and/or customary variables needed to build in Java

Name	Use
JAVATOP	Directory with **JDK** to use for compiling. Should be **JDK 1.2** or later compatible.
JAVA_CLASSES	Required list of Java classes to compile. Note these are class names including package prefix.
JAVA_JNICLASSES	Optional list of Java classes having JNI methods. Note these are class names including package prefix, as for **JAVA_CLASSES**.
JAVA_NMICLASSES	Optional list of Java classes having NMI methods. Note these are class names including package prefix, as for **JAVA_CLASSES**.
JAVA_BOOTCLASSPATHS	Variable with list of white-space-separated paths for option **-bootclasspath**. Required, unless **JAVA_EXTDIRS** is defined.
JAVA_EXTDIRS	List of white-space-separated paths for option **-extdirs**. Required, unless **JAVA_BOOTCLASSPATHS** is defined.
LOCAL_CLASSPATHS	List of white-space-separated directories/JARS/zips added to the classpath.

8. JNI files are generated into **$(JAVA_JNIDIR)** (which is by default **$(JAVA_INCDIR)/jni**)
9. NMI files are generated into **$(JAVA_NMIDIR)** (which is by default **$(JAVA_INCDIR)/nmi)**

Table A-1: Required and/or customary variables needed to build in Java <Helv9R>(Cont'd.)

Name	Use
JAVA_SOURCEPATHS	List of white-space-separated paths for option **-sourcepath**. Default: **src**
JAVA_REQUIRES	List of component names that this component requires to compile. Usual format: **<name>$(JAVA_REL_SUFFIX)....**
JAVA_CLASSES_WILDCARDS	List of **<directory!java_file>** separated with ! and containing the wildcard symbol. For example, if directory **aa** contains files **bb/cc.java** and **bb/dd.java**, then **JAVA_CLASSES_WILDCARDS = aa!*/*.java** results in the classes **bb.cc** and **bb.dd** being added to **JAVA_CLASSES**.

The following variables are optional variables, that usually:

- Do not need defining
- Have sensible defaults

Table A-1 lists the optional java makefile variables.

Table A-2: Optional Java makefile variables

Name	Use
JAVAC	Java compiler to use. Should be **JDK 1.2** or later. Currently cannot be **jikes**. Default: **$(JAVATOP)/bin/javac.exe**
JAVAH	Java header program. Default: as for **JAVAC**, but with **javah.exe**
JAVAJAR	Java JAR program. Default: as for **JAVAH**, but with **jar.exe**
LOCAL_JAVACFLAGS	Additional user options to pass to **$(JAVAC)**.
LOCAL_JAVAHFLAGS	Additional user options to pass to **$(JAVAH)**.
LOCAL_JAVAJARFLAGS	Additional user options to pass to **$(JAVAJAR)**.
JAVA_REL_SUFFIX	This is used to form the **JAVA_CLASSESDIR** and **JAVA_JARFILE** names. Default: **_g** for debug builds, empty otherwise.
JAVA_DIR_BUILD	Root directory for Java build results for component. Default is the same as the other results for the component: **$(_TMTGTBUILDROOT)/$(DIR_LOCAL)**
JAVA_TARGET	Base name used for resultant JAR file. Default: same as for object library result (last part of **DIR_LOCAL**).
JAVA_JARFILE	Resultant JAR file. Default: **jar/$(JAVA_TARGET)$(JAVA_REL_SUFFIX).jar** Inside the **comps/generated/lib** directory of the build.
JAVA_CLASSESDIR	Directory where .class files will be stored. Default: **$(JAVA_DIR_BUILD)/classes$(JAVA_REL_SUFFIX)**
JAVA_INCDIR	Directory where **jni** and **nmi** include directories will be created. Used in definition of **JAVA_JNIDIR** and **JAVA_NMIDIR**. Default: **$(JAVA_DIR_BUILD)/inc**
JAVA_JNIDIR	Directory where generated jni header files will be stored. Default: **$(JAVA_INCDIR)/jni**
JAVA_NMIDIR	Directory where generated nmi header files will be stored. Default: **$(JAVA_INCDIR)/nmi**

Table A-2: Optional Java makefile variables <Helv9R>(Cont'd.)

Name	Use
JAVA_DEBUG	Controls the use of javac -g option; this option is only used if this variable is defined. Default: -g:none if $(_TMTGTREL)==retail -g if $(_TMTGTREL)==debug otherwise empty [1]. Old options dbg and ret/release are also supported.
JAVA_TARGETVERSION	Value of option -target for $(JAVAC). Default: 1.1. The -target option is only used if this variable is not blank.
LOCAL_CLASSDEPENDS	Additional files that .class files are dependent on.

[1] If no -g option is given to javac, then line number and source file information are recorded, which is useful in an assert style build type.

A.2.8 Internal SDE2 variables

Table A-3 lists variables internal to SDE2.

Table A-3: Internal SDE2 variables

Name	Use
JAVA_COPTS	Real options passed to $(JAVAC). Created from LOCAL_JAVACFLAGS, plus options derived from JAVA_CLASSESDIR JAVA_BOOTCLASSPATHS JAVA_EXTDIRS LOCAL_CLASSPATH* JAVA_SOURCEPATHS JAVA_DEBUG
_SDE_CLASSDEPENDS	Contains files that SDE2 considers important enough to recompile all .class files. Default value: makefile and all .jar and .zip files referenced by JAVA_BOOTCLASSPATHS and JAVA_EXTDIRS

A.3 javac class and source file search mechanism

It is useful to repeat how javac searches for class and source files (this is from the Sun JDK 1.3 javac documentation):

> "When the compiler needs type information, it looks for a source file or class file which defines the type. The compiler searches first in the bootstrap and extension classes, then in the user class path. The user class path is defined by setting the CLASSPATH environment variable or by using the -classpath command line option. If you use the -sourcepath option, the compiler searches the indicated path for source files; otherwise the compiler searches the user class path both for class files and source files. You can specify different bootstrap or extension classes with the options -bootclasspath and -extdirs options.
>
> A successful type search may produce a class file, a source file, or both. Here is how javac handles each situation:
>
> Search produces a class file but no source file: javac uses the class file.
>
> Search produces a source file but no class file: javac compiles the source file and uses the resulting class file.

Search produces both a source file and a class file: **javac** determines whether the class file is out of date. If the class file is out of date, **javac** recompiles the source file and uses the updated class file. Otherwise, **javac** just uses the class file.

javac considers a class file out of date only if it is older than the source file. (The **-Xdepend** option specifies a slower but more reliable procedure.)

Note that **javac** can silently compile source files not mentioned on the command line. Use the **-verbose** option to trace automatic compilation."

Appendix B
Assembler Support

User Manual Version 3.8 Sep 29, 2006

What is covered in this appendix?

This appendix describes how to use assembler source files with SDE2.

B.1 Using assembler source files with SDE2

Assembler source files are located in the component **src** directory or its subdirectories. A list of their names has to be in the **S_SOURCES** makefile variable. **S_SOURCES** contains relative locations, names and extensions of all assembler files. The assembler files are treated in the same way as C/C++ source files. However, they are not reusable for different platforms. You can put them in the separate directories for each separate platform (like **tmComp7** for C files).

The assembler file extensions that are supported in SDE2 are:

.s, **.S**, **.asm**

The rule states that.S assembler files need to go through the C preprocessor before being assembled, but **.s** assembler files do not.

You can set your own compilation options in **_TMTGTAOPTS** and **TARGET_AFLAGS**. The order of the compilation options in the compiler line is:

<_SDE_AOPTS_FILE1><_SDE_AOPTS_FILE2><_SDE_AOPTS_FILE3><_TMTGTAOPTS><TARGET_AFLAGS>

The SDE2 stores assembler options in **<DIR_INTERM>/a.opt**. If the options not only depend on the component makefile (default), but also on other makefiles, you can set the list of all dependencies in **_SDE_AOPTFILE_DEPENDS**.

.

Appendix C
Visual Studio Integration

User Manual Version 3.8 **Sep 29, 2006**

What is covered in this appendix?

This appendix describes the extra actions needed to use Microsoft Developer Studio (MSDEV) in combination with DVP-SDE2, including:

- Setting up the component directory structure
- Starting a Visual C++ project
- Using Microsoft Developer Studio to build a DVP2 component
- Setting up the browse-info database

In Microsoft Visual C++, the Project Workspace is a container for your development projects. When you create a new project, a workspace is created at the same time. You use the Project Workspace window to view and access the various elements of your projects. This means that we use a project to generate/develop our code, and debug it for the WinNT versions.

C.1 Setting up your component's directory structure

The DVP-SDE2 has no predefined location for Visual C++ project files. We recommend you have a different specific project for each component you are developing. This suggests that the project's files should be located next to the component. The internal structure of the component's directory is extended with a Microsoft Developer Studio directory for the **tmComp11** component.

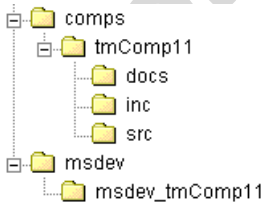


Figure 3-1: Directory structure for tmComp11 component

The **msdev_tmComp11** directory is meant to hold all files related to this component's Visual C++ project, including:

- Browse-info files for this component
- Visual C++ project administration files
- All scripts and temporary files to use the Developer Studio capabilities.

Note that the **msdev_tmComp11** directory is not part of the DVP component release, CMSynergy™ might however put it under version control.

Complete the following steps to set up your component's directory structure.

1. Create the directory structure for your new component, including the **msdev_<comp-name>** directory next to the component.
2. Copy the following batch and script files from the **<msdev/msdev_tmComp11>** example directory to your newly created **msdev_<comp-name>** directory: **make_comp.bat, make_bsc.bat**
3. Change the read-only attribute of the following batch file: **<your SDE2_Root>/project/sites/blrsdm/x86_nt_debug_static_default.bat** to make it writable.
4. Tailoring of these files is explained in *Section C.3, Building the DVP2 component from within the Developer Studio* on page 127 and will be addressed later.

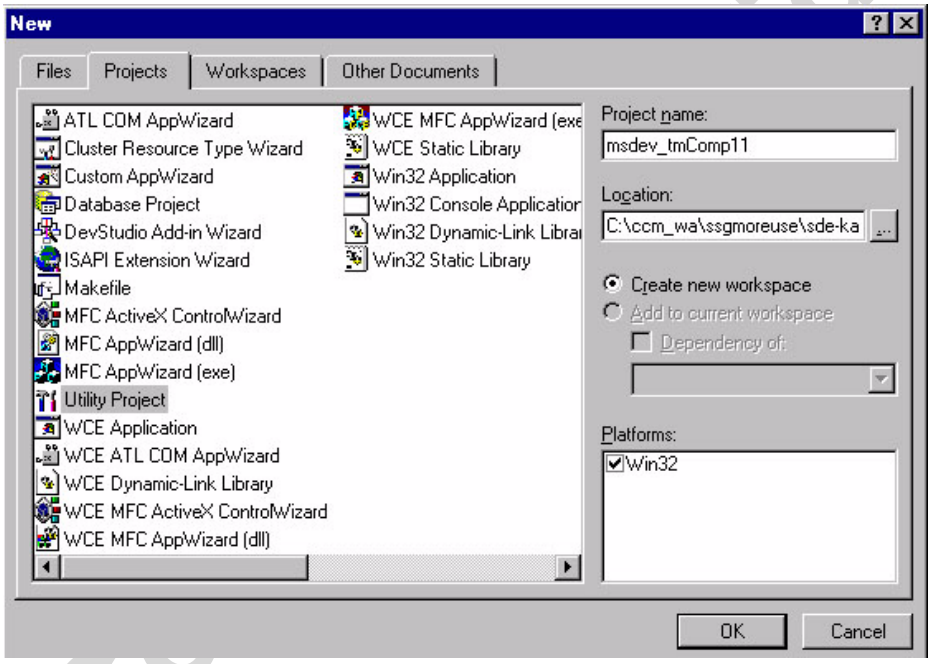


Figure C-2: Create the msdev_tmComp11 project

C.2 Starting the component's Visual C++ project

Each project has a project type, which you choose when you create the project. The project type specifies what to generate and also specifies some default settings required in order to build that output type.

Because we don't want to build any Microsoft-specific applications we don't want Visual C++ to add any predefined libraries. To do so, we use the utility project, in which no files are added to the project. The utility project does not generate any predetermined output files, such as a **.lib**,**.dll** or **.exe**.

Complete the following step to create your MSDEV project:

1. Start MSDEV.

2. Select **File** → **New** → **Project**.

   Now you see a dialog window.

3. Select **Utility Project**, Project name: **msdev_<your-comp>**, Location: **<your SDE2 path/sde_template/msdev>**. Select **Create new workspace**, mark **Win32 Platform**.

4. Click **OK**.

5. Click **OK** again to finish creating your project.

Complete the following steps to add files to your project.

1. Select **Project "Add to project" Files**.

2. Select your component's files; both ***.c** and ***.h**.

   Do not add external header files.

3. Click **OK**.

   Now your files are visible in MSDEV's file-view.

C.3 Building the DVP2 component from within the Developer Studio

The DVP-SDE2 uses a proprietary build process that does not comply with the make-structure of MSDEV. The DVP-SDE2 build process is command-line based and cannot be called from within MSDEV using the **build** command.

The DVP2 build process can be automated using script files:

- **<make_comp.bat>**[10] – Needs minimal tailoring per component

and[11]:

- **<x86_nt_debug_static_default.bat>** – Needs tailoring
- **<make_comp.bat>** – Needs no tailoring
- **<make_comp_bsc.bat>** – Needs no tailoring
- **<parse.pl>** – Needs no tailoring

which are called from within MSDEV by customized tools.

Now make the necessary changes to the following files to tailor the files to your needs:

- **<make_comp.bat>** in **<your SDE2_ROOT>/msdev/msdev_<your_comp>**
- **<x86_nt_debug_static_default.bat>** in **<your SDE2_ROOT>/project/sites/blrsdm**

C.3.1 Customizing MSDEV to call the build scripts

MSDEV allows you to customize the **Tools** menu by adding, editing, and deleting menu items. You can add frequently used utilities to the **Tools** menu and run them from within MSDEV. We use this option to add a tool calling **<make_comp.bat>**.

Complete the following steps to add the DVP2 make tool:

10. This file is located in **<your DVP2_ROOT>/msdev/msdev_<your_comp>**
11. These files are located in **<your DVP2_ROOT>/project/sites/blrsdm**

1. From the **Tools** menu, click **Customize**, and then click the **Tools** tab.
2. In the **Menu Contents** box, scroll to the bottom of the list, double-click the blank line (indicated by an empty rectangle), and type the name of the tool, for example, **Make DVP2 component**.
3. Press **Enter**.
4. In the **Command** field, type:

   <Your SDE2_Root>\project\sites\blrsdm\make_comp.bat
5. In the **Arguments** field, type:

   $(WkspDir)

4. In the **Initial Directory,** type:

   $(WkspDir)
6. Click to select the **Use Output Window** check box.

   Now you should see a window like the one shown in Figure C-1.
7. Click **Close**.

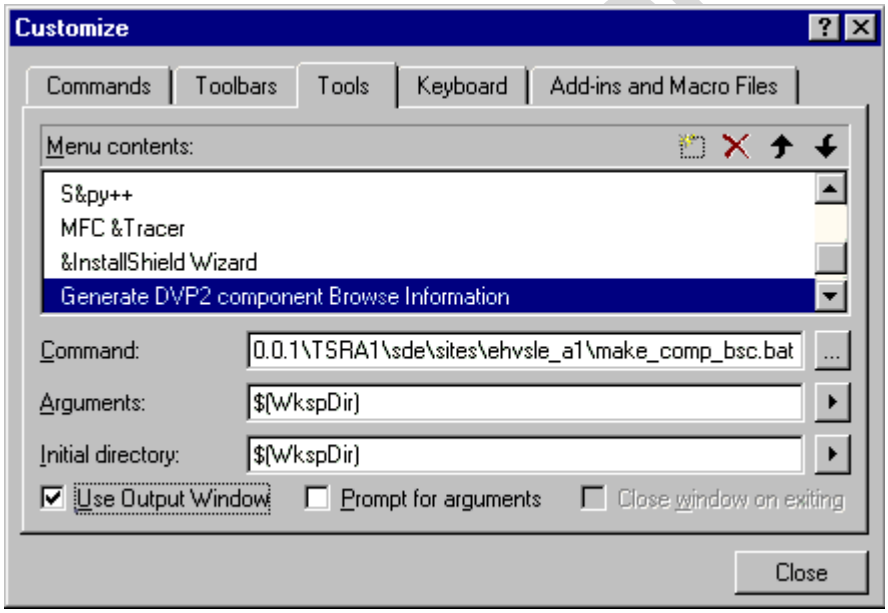


Figure C-1: Adding the DVP2 make tool

The DVP(-SDE)2 make tool can be called from the **Tools** menu. When called, the tool will not ask for arguments. The result will be the start of the component's build, and the output will be seen in the MSDEV output window.

C.3.1 Error parsing

Because you selected the **Use Output Window** check box for the **Make DVP(-SDE)2 component tool** on the **Tools** tab of the **Customize** dialog box (see Figure C-1), the Output window's error parser interprets the output of the tool. You can jump to source code syntax errors (and warnings) directly from the error list in the Output window.

To do so, the errors generated by the 2 make tool are caught and parsed by the <parse.pl> perl script, and the formatted errors/warnings are forwarded to MSDEV. No special actions are necessary to arrange this.

C.4 Using code browse information

MSDEV uses a browse information database to hold the reference information of all source files within a project. Therefore, to make use of the browsing capabilities of MSDEV you first need to build the browse information database.

Both DVP1-SDE and DVP-SDE2 have a proprietary build process, which does not comply with the make structure of MSDEV. Therefore the MSDEV internal generation of code-browse information (which depends highly on the make structure) cannot be used. The code-browse information can however be generated from the command line, and can then be used in MSDEV.

Because DVP1-SDE and DVP-SDE2 have different build structures, the code-browse information for MSDEV is generated in different ways.

C.4.1 Building DVP1 component browse information

All TSR-A1 development is done using the DVP-SDE2; as a result, there are no changes in the existing DVP1 tree. The code-browse information database for the DVP1 tree has to be built just once, and the generated <DVP1.bsc> file can be used in MSDEV (e.g., to study the existing DVP1 code).

C.4.2 Building DVP2 component browse information

Code-browse information for your DVP2 component (i.e., your project's component) is generated in two phases:

1. When building for Pentium-WinNT, for each <source.c> file, a corresponding <source.sbr> file is generated in the **<msdev_<comp_name>>** directory. This is automatically done when building for WinNT provided that you add a local compile flag for your WinNT in your local makefile.

   Add the compile flag by adding the following custom options to your makefile.

```
#----------------------------------------------------------
# Options for the C compiler
#----------------------------------------------------------
_TMTGTCOPTS= -Fr
```

   The generated.sbr files are now in your **<msdev_<comp_name>>** directory, and will be collected in one component's browse-info file <comp_name>.bsc using the <make_bsc.bat> MS-DOS batch file.

2. Tailor the DVP2_ROOT_DIR and COMP_NAME variables in <Your DVP2_ROOT>/msdev/msdev_<your_comp>/make_bsc.bat

C.4.3 Using DVP2 component browse information

To call the <make_bsc.bat> file from within MSDEV, you have to add a **tool** in MSDEV, in the same way the DVP2 make tool was created:

Complete the following steps to add the **Generate DVP2 component browse info** tool.

1. On the **Tools** menu, select **Customize**, and then click the **Tools** tab.
2. In the **Menu Contents** box, scroll to the bottom of the list, double-click the blank line (indicated by an empty rectangle), type the name of the tool, for example **Generate DVP2 component browse info**.
3. Press **Enter.**
4. In the **Command** field type:

   <DVP2_Root>\sde\sites\ehvsle_a1\make_comp_bsc.bat
5. In the **Arguments** field type:

   $(WkspDir)
6. In the **Initial directory** field type:

   $(WkspDir)
7. Click to select the **Use Output Window** check box.

   Now you should see a window like the one shown in Figure C-2.
8. Click **Close**.

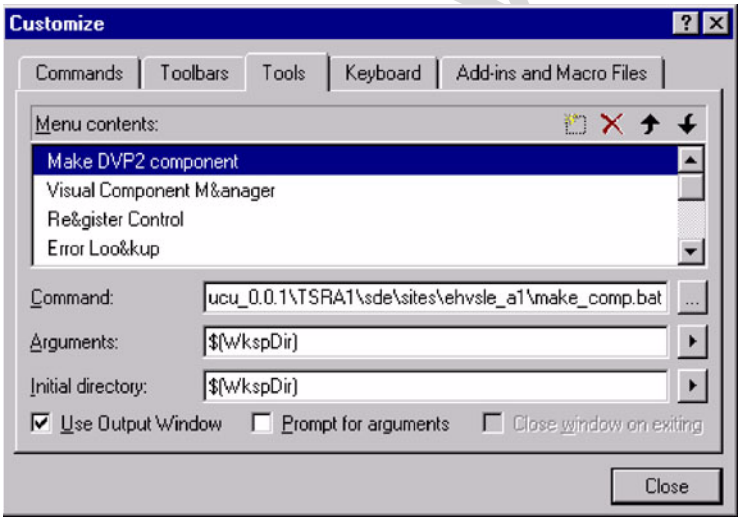


Figure C-2: Add tool to generate DVP2 component browse info

The Generate Browse Information tool is called from the Tools menu. When called, the tool does not ask for arguments. The result is the start of the collection of the component's browse-info files, and the output will be seen in the MSDEV output window.

Note: When generating a new version of your component's browse-info file, be sure to close the browse-info file in MSDEV (using Alt-T-F).

When the component's browse-info file is successfully created, a file named <comp_name.bsc> exists in your **<msdev_<comp_name>>** directory.

Complete the following steps to load the component browse-info into MSDEV.

1. Select **File** → **Open.**
2. Browse to your generated <comp_name.bsc> file, and select **Open.**

Appendix D
Autodocumentation

User Manual Version 3.8 **Sep 29, 2006**

What is covered in this appendix?

This appendix contains an overview of the document generating tool Doxygen, as well as instructions for using Doxygen with SDE2.

D.1 Doxygen overview

SDE2 supports the generation of documentation from source code with the tool Doxygen. The currently supported version is 1.4.3.

This tool is a freeware tool that is delivered with the PC distribution of SDE2. For Unix, the tool can be downloaded from **http://www.doxygen.org .**

Autodocumentation only makes sense when the source code contains Doxygen tags and comments. More information about these tags and their use, and features of doxygen can be found in **sde/autodoc/docs/14_user_doc .** Doxygen has built-in support to generate inheritance diagrams for C/C++/Java classes. Doxygen can use the "dot" tool from graphviz 1.16 to generate more advanced diagrams and graphs. Graphviz is an open-source, cross-platform graph drawing toolkit from AT&T and Lucent Bell Labs and can be found at **http://www.research.att.com/sw/tools/graphviz .**

Note: SDE2 does not include **hhc.exe**. This file is delivered by Microsoft and may not be redistributed. However, you can downloaded it from:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/htmlhelp/html/vsconHH1Start.asp
Put it in your **`/sde_template/sde/autodoc`** directory. If you deliver SDE2 to external customers, remove this file from SDE2 tree.

The following information is given by Dimitri van Heesch, the author of Doxygen:

> “All doxygen does is generate a HTML help project file. Doxygen's license explicitly permits a user to use the output generated by doxygen for any purpose, so also in combination with non-GPLed tools (like **hhc.exe**). By setting the **GENERATE_TREEVIEW** option to **YES** in the config file, Doxygen can also generate a HTML help a-like output in plain HTML + Javascript”.

D.2 Creating documentation from the component directory

SDE2 supports the generation of both user documentation and design documentation in HTML,RTF, PDF and LaTeX formats. The type of documentation to be generated can be controlled by the environment variable **_TMDOC**. The values for this environment variable is comma separated, the possible values are html, rtf, pdf and tex. Eg: If **_TMDOC**=pdf,html only PDF and HTML documents will be generated. The differences are found in the level of detail. User documentation is extracted from the public header files of a component. Design documentation is extracted from all source and header files of a component.

SDE2 provides information about the required components with hyperlinks, required libraries with their diversity flavors (only for design docs) and whether the component is required in debug, assert or retail version. All hyperlinks between different component documentation are made based on **REQUIRES**, **JAVA_REQUIRES**, **_SDE_IMPORT_LIBS**, **_SDE_IMPORT_DLLS**[12] and **_<ComponentName>_SUFFIX** variables in the makefile/SDE2.

D.2.1 User documentation

User documentation of a component is generated by going to the component's directory and typing:

gmake userdoc

The result is the directory **$(_TMTGTBUILDROOT)/comps/<component>/docs/14_user_doc/html** containing HTML files. The entry point is **index.html** for the HTML generation.

$(_TMTGTBUILDROOT)/comps/<component>/docs/14_user_doc/rtf containing RTF file

$(_TMTGTBUILDROOT)/comps/<component>/docs/14_user_doc/pdf containing PDF file

$(_TMTGTBUILDROOT)/comps/<component>/docs/14_user_doc/latex containing LaTeX files

D.2.2 Design documentation

Design documentation of a component is generated by going to the component's directory and typing:

gmake devdoc

The result is the directory **$(_TMTGTBUILDROOT)/<components>/tmComp1/docs/03_arch_dsgn/html** containing html files. The entry point is **index.html**.

$(_TMTGTBUILDROOT)/comps/<component>/docs/03_arch_dsgn/rtf containing RTF file

$(_TMTGTBUILDROOT)/comps/<component>/docs/03_arch_dsgn/pdf containing PDF file

$(_TMTGTBUILDROOT)/comps/<component>/docs/03_arch_dsgn/latex containing LaTeX files

In both cases, your component makefile should include (indirectly via **makelib.mk** or **maketarget.mk) common.mk**. If not, you have to include **autodoc.mk** in your makefile.

For generating PDF documents epstopdf and pdflatex should be installed on the system and also defined the _EPSTOPDF and _PDFLATEX environment variables to absolute path of these executables.

12. **_SDE_IMPORT_DLLS** depends on whether DLL is already generated, so it may be the case that the component generates a DLL, but this component is mentioned in **_SDE_IMPORT_LIBS** instead of **_SDE_IMPORT_DLLS**.

eg: **set _EPSTOPDF=c:/epstopdf/epstopdf.exe**

set _PDFLATEX=c:/miktex/bin/pdflatex.exe

D.2.3 User configurable Auto Documentation variables

The following variables can be configured by setting these variables in either the environment or the component/application makefile. If not set, SDE2 would set these variables to default values. For more details on the definistions of the variables refer the User Guide/Help of Doxygen located in <_TMROOT>/sde/autodoc/docs/09_usr_doc/ directory (filename:**doxygen_manual.chm**).

DOC_VERSION

DOC_STATUS

DOC_AUTHOR

DOC_LOGO

DOC_COMPNAME

DOC_COMP_DIR

DOC_FILES2COPY

DOC_FILES2COPY_user

DOC_FILES2COPY_dev

DOC_doxyfile

DOC_PROJECT_NAME

DOC_PROJECT_NUMBER

DOC_OUTPUT_LANGUAGE

DOC_OUTPUT_DIRECTORY_user

DOC_OUTPUT_DIRECTORY_dev

DOC_OUTPUT_DIRECTORY

DOC_DISABLE_INDEX

DOC_EXTRACT_ALL_user

DOC_EXTRACT_ALL_dev

DOC_EXTRACT_ALL

DOC_EXTRACT_PRIVATE_user

DOC_EXTRACT_PRIVATE_dev

DOC_EXTRACT_PRIVATE

DOC_HIDE_UNDOC_MEMBERS

DOC_HIDE_UNDOC_CLASSES

DOC_BRIEF_MEMBER_DESC

DOC_REPEAT_BRIEF

DOC_ALWAYS_DETAILED_SEC

DOC_FULL_PATH_NAMES

DOC_STRIP_FROM_PATH

DOC_INTERNAL_DOCS_user

DOC_INTERNAL_DOCS_dev

DOC_INTERNAL_DOCS

DOC_CLASS_DIAGRAMS

DOC_SOURCE_BROWSER_user

DOC_SOURCE_BROWSER_dev

DOC_SOURCE_BROWSER

DOC_INLINE_SOURCES

DOC_STRIP_CODE_COMMENTS

DOC_CASE_SENSE_NAMES

DOC_VERBATIM_HEADERS

DOC_SHOW_INCLUDE_FILES

DOC_JAVADOC_AUTOBRIEF

DOC_INHERIT_DOCS

DOC_INLINE_INFO

DOC_SORT_MEMBER_DOCS

DOC_TAB_SIZE

DOC_ENABLED_SECTIONS

DOC_QUIET

DOC_WARNINGS

DOC_WARN_IF_UNDOCUMENTED

DOC_INPUT_user

DOC_INPUT_dev

DOC_INPUT

DOC_FILE_PATTERNS_user

DOC_FILE_PATTERNS_dev
DOC_FILE_PATTERNS

DOC_RECURSIVE_user

DOC_RECURSIVE_dev

DOC_RECURSIVE

DOC_EXCLUDE

DOC_EXCLUDE_PATTERNS

DOC_EXAMPLE_PATH

DOC_EXAMPLE_PATTERNS

DOC_IMAGE_PATH

DOC_INPUT_FILTER

DOC_ALPHABETICAL_INDEX

DOC_COLS_IN_ALPHA_INDEX

DOC_IGNORE_PREFIX

DOC_GENERATE_HTML

DOC_HTML_OUTPUT

DOC_HTML_STYLESHEET

DOC_HTML_ALIGN_MEMBERS

DOC_GENERATE_HTMLHELP

DOC_HTML_HEADER_TPL_user

DOC_HTML_HEADER_TPL_dev

DOC_HTML_HEADER_TPL

DOC_HTML_FOOTER_TPL_user

DOC_HTML_FOOTER_TPL_dev

DOC_HTML_FOOTER_TPL

DOC_GENERATE_LATEX

DOC_LATEX_OUTPUT

DOC_COMPACT_LATEX

DOC_PAPER_TYPE

DOC_EXTRA_PACKAGES

DOC_PDF_HYPERLINKS

DOC_LATEX_BATCHMODE

DOC_LATEX_HEADER_TPL_user

DOC_LATEX_HEADER_TPL_dev

DOC_LATEX_HEADER_TPL

DOC_GENERATE_RTF
DOC_RTF_OUTPUT

DOC_COMPACT_RTF

DOC_RTF_HYPERLINKS

DOC_GENERATE_PDF
DOC_PDF_HYPERLINKS

DOC_GENERATE_MAN

DOC_MAN_OUTPUT

DOC_MAN_EXTENSION

DOC_ENABLE_PREPROCESSING

DOC_MACRO_EXPANSION

DOC_SEARCH_INCLUDES

DOC_INCLUDE_PATH

DOC_PREDEFINED

DOC_EXPAND_ONLY_PREDEF

DOC_TAGFILES

DOC_GENERATE_TAGFILE

DOC_ALLEXTERNALS

DOC_PERL_PATH

DOC_CLASS_GRAPH

DOC_COLLABORATION_GRAPH

DOC_INCLUDE_GRAPH

DOC_GRAPHICAL_HIERARCHY

DOC_SEARCHENGINE

DOC_CGI_NAME

DOC_CGI_URL

DOC_DOC_URL

DOC_DOC_ABSPATH

DOC_BIN_ABSPATH

DOC_EXT_DOC_PATHS

Appendix E
Autodocumentation - Docjet

User Manual Version 3.8 **Sep 29, 2006**

What is covered in this appendix?

This appendix contains an overview of the document generating tool Docjet, as well as instructions for using Docjet with SDE2.

E.1 Docjet overview

SDE2 supports the generation of documentation from source code with the tool Docjet. The currently supported version is 5.2.

This tool is a free to try tool that can be downloaded from **http://www.tall-tree.com/sdk.php4**

Autodocumentation only makes sense when the source code contains Docjet tags and comments.

SDE2 does not include **htmlhelp.exe**. This file is delivered by Microsoft and may not be redistributed. However, you can download it from:
http://msdn.microsoft.com/library/default.asp?URL=/library/tools/htmlhelp/chm/hh1start.htm
and install it on your local system.

E.2 Creating documentation from the component directory

SDE2 supports the generation documentation HTML and Microcoft Help (.cfm) formats. User documentation.

DOCJET_HOME environment variable needs to be defined and set to the installation location of Docjet software.

For eg:

set DOCJET_HOME=d:/progra~1/talltree/Docjet

Documentation of a component can be generated by running the command in the component directory:

gmake docjet

Documents will be generated in the component directory under the **Documentation/** directory.

Docjet uses the **config.djt** located in the **$_TMROOT/sde/docjet/** directory for generating documents. If one needs to use a user defined configuration file then the **DOCJET_CONFIG** environment variable needs to be defined to the location of the configuration file.

For eg:

set DOCJET_CONFIG=d:/sde2.djt

Appendix F
Make Utilities and Cygwin

User Manual Version 3.8 **Sep 29, 2006**

What is covered in this appendix?

This appendix background regarding different versions of GNU make and Cygwin, and when it is appropriate to use which version.

F.1 GNU make utility

Only one GNU **make** utility is supported since the release of SDE2 1.2 on a PC. It is the standard GNU **make** version 3.80 utility compiled for Windows 32. Note that there exists a GNU **make** utility compiled for i686-cygwin, which does not correctly interpret **VPATH**. Do not use this release. On Unix/Linux GNU **make** is a standard utility; use version 3.80.

NOTE: SDE2 doesnot work with GNU make 3.81

F.2 Cygwin utility

Additionally, we use the Cygwin toolset version 1.3.3 on a PC. You can download the newest version from **www.cygwin.com**. If you need to have some Cygwin executables, different from the provided ones, submit a CR for this.

Note that in SDE2 1.1.6 we used Cygwin version 1.3.4. However, that version was slow and therefore we moved back to version 1.3.3.

Different problems have been encountered with the Cygwin toolsets. They were mainly related to the speed performance under WinNT. The SDE2 team is looking at the usage of the MKS (**www.mks.com**) toolset, which performs better (currently on version 8.0). However, there is no guarantee for backward compatibility. It is not shareware software and thus it cannot be distributed with SDE2. More information about the alternative toolsets can be found in *SDE2UnixUtilityAnalysis.pdf* document.This document is located in the **`sde/docs`** directory.

SDE2 1.2 is faster with respect to SDE2 1.1.6. This is due to the fact that lot of sed and awk scripts are replaced by GNU **make** scripts. Some awk and sed scripts still remain, but they are likely to be replaced in a future SDE2 release.

SDE2 1.5 is faster with respect to SDE2 1.2 because one more sed script is removed, but the improvement is small. Also, you could speed up your system removing some add-on features, see *Section 5.2.7, Tuning sde directory* on page 112.

F.3 Shells

One interesting tip is using a **SHELL** variable. According to GNU, there is the following search order for shell utilities:

1. In the precise place pointed to by the value of **SHELL**. For example, if the makefile specifies **SHELL = /bin/sh**, **gmake** will look in the directory **/bin** on the current drive.
2. In the current directory.
3. In each of the directories in the **PATH** variable, in order.

We recommend setting the **SHELL** variable to one of the shells.

F.4 Network

An important issue with respect to the Cygwin speed performance is that you should avoid using network drives if possible. If you have a Cygwin distribution on your network drive, move it to your local drive.

F.5 Implicit rules

The normal way of specifying rules for GNU **make** is by using the mechanism of implicit rules. Usage of implicit rules by the GNU **make** utility can be suppressed with the **-r** option. For example, set this for your build scripts as:

```
MAKE = gmake -r
```

or

```
MAKE = gmake -r --no-print-directory
```

It is recommended to use this option, because quite often implicit rules are not used by GNU **make**. If you suppress them, it speeds up the make process.

However, for some file extensions, rules might be needed; this can be done using suffix rules defined in the makefile. This is done in the following way:

```
.SUFFIXES:
.SUFFIXES: .o .c .cpp
```

Appendix G
Concept of SDE2 (makefile) structure

User Manual Version 3.8 Sep 29, 2006

What is covered in this appendix?

This appendix provides information about the makefile directory structure of SDE2, and variables that can be used with the **maketarget<_TMBSL>.mk** file.

G.1 SDE2 organization

Building a component requires the following information

- Component information
  - Source files
  - Dependencies on other components
- Project- and configuration-specific information
  - The compiler environment (this is a mix of CPU Class and OS class, for example TriMedia-pSOS suite, or MIPS-WinCE suite)
  - The CPU (R3940, tm32, etc.)
  - The endianness
  - The release type (debug/retail/assert)
  - The link type (static/dynamic)
  - The location of BSPs, OS files, standard library

The component information is unique for each component and should be part of the component's makefile. The component's makefile can be found in the main directory of the component, for example, **`comps/tmComp1/makefile`**

The configuration-specific information should not be part of the component makefile, because it makes the component less reusable in another environment. Therefore some generic makefiles are defined in the **sde** directory according to the following structure:

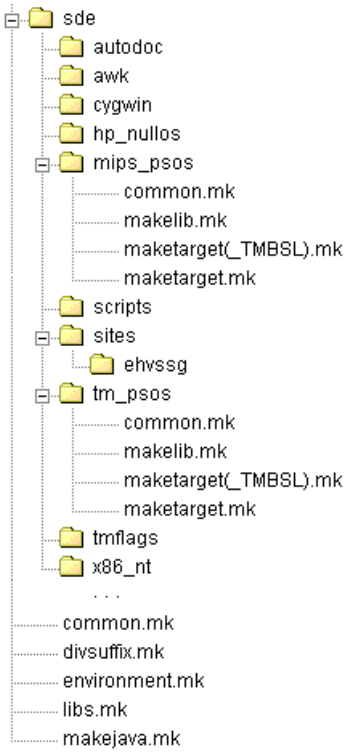


Figure 7-1: SDE directory structure

Here we describe the information that is contained in these files and directories.

environment.mk – The entry point for component-specific makefiles. This file computes paths that are configuration class–specific. It sets important makefile variables. It is the first file you invoke in your component makefile after the component name setting.

common.mk – Used for computing dependencies and defining the rules for making objects from C/C++/ASM files. Computes the locations of generated files.

makejava.mk – Used for Java compilation. It creates intermediate class file(s) and final JAR file. The JAR file is placed in <**_TMTGTBIUILDROOT>/comps/generated/lib/jar**.

divsuffix.mk – Checks whether a valid release type has been chosen and sets the correct compilation options with respect to **_TMTGTREL**.

libs.mk – Contains rules to determine all recursively required libraries, DLLs and their suffixes.

awk directory – Contains the awk scripts used in SDE2.

scripts directory – Contains the build and test scripts for SDE2.

tmflags directory – Contains the files to generate **tmFlags.h**, **tmFlags.mk** and **tmFlags.cfg** (see *Section 3.3, Standard precompile flags and tmFlags.h file* on page 29).

autodoc directory – Contains makefiles for generating documentation from the source files of a component, see also *Appendix D*.

cygwin directory – Contains tools required by SDE2 when building on a PC host platform.

The remaining directories (**mips_psos**, **tm_psos**, **hp_nullos**, **x86_nt**, **arm_nullos**, **arm_vxworks**, **x86_ce**, **mips_ce**, etc.) are configuration class–specific directories. For each configuration class a directory is specified containing three or more makefiles:

- **common.mk**
- **makelib.mk**
- **maketarget.mk**
- **maketarget<_TMBSL>.mk**

The **common.mk** file contains the compile and link options for the specific configuration class. The **makelib.mk** file contains the rule to archive objects to a library or DLL. The **maketarget<_TMBSL>.mk** files contain the rules to link objects of the test application, a set of libraries (defined by the test application's makefile), and platform-specific libraries, objects and settings to an executable. You can have one or more maketarget files.

The **sde** directory with its makefiles provides functionality that is used by the makefiles of components. The result is that the makefiles of the components are easy to make, understand, and use, while the makefiles of the **sde** directory contain all the complexity.

It is important to realize that the makefiles of the **sde** directory should be the same across all NXP Semiconductors (NXP) development sites. The interface of the SDE makefiles to the component's makefiles may not be modified, only extended. The extensions must be implemented in a controlled way, such that the whole NXP community has access to the latest status of the makefiles of the **sde** directory. In this way, you can guarantee that components developed at site A also work at site B. When development sites have new requirements for the SDE2, they must submit a change request to the MoReUse SoCDT/LIPP department, see *[RULES] MoReUse Rules Document* listed in the Bibliography.

G.2 The maketarget<_TMBSL>.mk files

Within a configuration class (for example **mips_psos**), different projects may have different ways to build their executables. Differences are found in the board support package, linking extra standard libraries (like pna), and so on. For that, development sites can make their own **maketarget<_bsl>.mk** file and put it in the **sde/<configuration-class>** directory (for example **sde/mips_psos/maketarget_newboard.mk**). By setting the environment variable **_TMBSL** to <_tmbsl> (example **_TMBSL=newboard**), SDE2 will use this makefile when executables are built for the corresponding configuration class.

The content of the new **maketarget<_bsl>.mk** makefile can be based on the content of an existing **maketarget<_bsl>.mk**. When makefiles have a lot in common, the common part can be put in a **maketarget_generic.mk** makefile, which is included by the **maketarget<_TMBSL>.mk**. For an example, see the makefiles of the **sde/mips_psos** directory.

Table G-1 shows the relevant SDE2 lines/variables for the new makefile.

Table G-1: The SDE2 variables that are relevant for maketarget<bsl>.mk

Variable	Description
_SDE_BUILD_TARGET	Must equal **EXE**. This variable is input for **$(DIR_SDE)/$(DIR_CONFIG)/common.mk**.The makefile must start with the definition of this variable
DIR_SDE	The path to the **sde** directory.
DIR_CONFIG	The directory containing the makefiles for the current configuration class.
$(DIR_SDE)/$(DIR_CONFIG)/common.mk	Location of the configuration class–specific **common.mk** makefile. This makefile must be included almost at the beginning of the **maketarget<_bsl>.mk**, right behind the definition of **_SDE_BUILD_TARGET**
_SDE_DIR_BIN	Location of the executable; computed by: **$(DIR_SDE)/$(DIR_CONFIG)/common.mk**
TARGET	Name of the executable (determined by makefile of executable)
_SDE_OPTIONFILES	The exact location and names of files containing the options for the compiler.
_SDE_OBJECTS	The exact location and names of the objects involved in the test executable.
_SDE_DEP_LIBRARIES	The exact location and names of the generic libraries this test executable depends on. This excludes libraries that are environment-specific (like **pSOS** libraries).

Appendix H
(PC)Lint support in SDE2

User Manual Version 3.8 **Sep 29, 2006**

What is covered in this appendix?

This appendix describes about the PC-Lint support in SDE2.

H.1 Lint Support in SDE2

SDE2 now allows the user to run PC-Lint, a static code analysis tool on all source files of a particular component. The user needs to install PC-Lint in his host PC and set the following environment variables:

LINTHOME, is set to the location, where PC-Lint has been installed (i.e. **lint-nt.exe**)

LINTFILE, is set to the configuration file for the pc lint with the .lnt extension
(eg: **CastorSW.lnt**)

PC-Lint can be invoked in SDE2 by typing **gmake lint**, on any component.

Appendix I
Adding a Configuration Class

User Manual Version 3.8 **Sep 29, 2006**

What is covered in this appendix?

This appendix describes how to add a configuration class.

I.1 Adding a configuration class

Adding a configuration class means adding a new tool set to the SDE2, for example PalmOS or **mips_nullos** tool set. Adding a configuration class to a local configuration can be done by local sites. However, the local site should submit a CR to the SDE2 development team and they will add it to the SDE2 as user-supplied feature.

Adding a configuration class requires the following steps:

- Identify the configuration class in terms of **_TMTGTOSCLASS**, **_TMTOOLCHAIN** and **_TMTGTCPUCLASS**. The default value of **_TMTOOLCHAIN** is undefined and you need to define it only if you use more than one toolchain.
- Create the **sde/< _TMTGTCPUCLASS>_<_TMTGTOSCLASS>** directory.
- Copy the **common.mk**, **makelib.mk** and **maketarget.mk** files of the **sde/tm_psos** directory to the **sde/<_TMTGTCPUCLASS>< _TMTOOLCHAIN>_<_TMTGTOSCLASS>** directory and tune them line by line in such a way that it fits your new tool set. Introduce as few as possible new environment variables.
- Create an initialization script in the **project/sites/blrsdm** directory. Take **tm_psos_debug_static_el_tm32_winnt_default.bat** as a starting point for this new file.
- Extend the **sde/tmflags/tmFlagsMak.mk** with new definitions for your tool set.
- Extend the **sde/common.mk**, search for the block "Configuration check (passive part)"

Appendix J
QAC

User Manual Version 3.8 **Sep 29, 2006**

What is covered in this appendix?

This appendix contains an overview of the tool QAC, as well as instructions for using QAC with SDE2.

J.1 QAC overview

QAC is a deep-flow static analysis tool which will increase productivity and improve quality standards in a C language development environment.

QAC is designed to identify problems in C source code that arise from language usage that is dangerous, over complex, non-portable, hard to maintain or which simply diverge from local coding guidelines.It will warn about many issues that are not reported by compilers or other development tools.

QAC will significantly reduce the time that needs to be spent inconducting code-reviews and will raise programmer awareness of features of the C language which are often not fully understood. By drawing attention to problems at an early stage in the development process, code quality will be improved and the testing cycle will be shortened.

J.2 QAC and SDE2

SDE2 has an integration for QAC. To use QAC with SDE2 the following environment variables need to be set

QACBIN - Path to the binary directory of QAC installation

QACPERSONALITIES-The compiler personality represents the characteristics of your compiler

QACPROFILE-Project profile

J.2.1 Running qac on components

To run qac on a component, following are the commands:

gmake qac

gmake qacref

gmake qacdiff

J.2.1.1 gmake qac

To run qac one needs change directory to the component directory and run **gmake qac**. The files generated by QAC (or the results) are stored in the following directory

```
$(_TMTGTBUILDROOT)/comps/<compname>/tmp/$(_SDE_LIB_CONFIGURATION)/qac
```

J.2.1.2 gmake qacref

The command **gmake qacref** is to store the results of QAC in a separate directory for further reference. The location of the directory is determined by the environment variable **QAC_REF_BASE**.

J.2.1.3 gmake qacdiff

The command gmake qacdiff is to run qac on the component and also to verify the results with that of the reference results earlier stored by the command gmake qacref. The location of the reference directory is determined by the environment variable **QAC_REF_BASE.**

In order to run this command "**diff**" utility needs to be installed on the windows system and should be in PATH

J.2.2 Running QAC on selected header files

SDE2 has the facility to run QAC on selected header files. To achieve this one needs to set the variable (environment or makefile variable) **QAC_HEADERS**.

QAC_HEADERS contains space separated values of the header files on which QAC should be run. These values can have path of the header file either relative from the existing directory or absolute path

The files generated by QAC (or the results) are stored in the following directory

```
$(_TMTGTBUILDROOT)/comps/<compname>/tmp/$(_SDE_LIB_CONFIGURATION)/qac
/headers
```

.

Appendix K
Eclipse Integration

User Manual Version 3.8 **Sep 29, 2006**

What is covered in this appendix?

This appendix contains an overview of eclipse, its integration with the SDE2 as a plug-in and usage of SDE2 with the Eclipse IDE.

K.1 Eclipse overview

Eclipse is an open platform for tool integration built by an open community of tool providers. It operates with a common public license that provides royalty free source code and world wide redistribution rights. It provides tool developers with ultimate flexibility and control over their software technology.

Eclipse is an integrated development environment for anything and for nothing in particular. The framework is written in Java, which makes it platform independent. It is available under Windows and Linux

Refer www.eclipse.org for more information.

K.2 Eclipse and SDE2

SDE2 is integrated to Eclipse framework through an independent plug-in. This SDE2 plug-in provides all existing SDE2 user interfaces by means of:

- Context sensitive drop-down menus (on component/application makefiles)
- Main menus for executing SDE2 scripts and setting scripts options
- Toolbar options to build/clean and run last command executed.

SDE2 user can also use all existing features of Eclipse framework like effectively writing and debugging source codes in classical programming languages (like C/C++, Java), editors with features like syntax highlighting, code-completion, views to show structure of code, source code navigation, etc.

K.2.1 Installation of SDE2 and Eclipse plugin

The following packages are required for using SDE2 within the Eclipse framework:

- **Eclipse Version 3.1** installation packages are available in http://www.eclipse.org/downloads/index.php.

Different packages of Eclipse are available for Windows, Linux (x86/Motif), Linux (x86/GTK 2) and Linux (IA 64/GTK 2)

Note : We recommend the Linux users to use Linux (x86/GTK 2) version of Eclipse installation.

- **C/C++ IDE development Tool Plug-in (CDT) Version 3.0.0** available in http://download.eclipse.org/tools/cdt/releases/eclipse3.1/dist/3.0.0/
- It can also be downlaoded from http://www.eclipse.org/downloads/index.php

Different packages of CDT are available for Windows and Linux.

- SDE2 plug-in, released by the SDE2 development team is available on **CODS (**http://pww.dtg.sc.philips.com/cods/searchItems.aspx**)**

This SDE2 plug-in is platform independent and works on Windows and Linux platforms.

K.2.1.1 Installation and working Procedure

4. Download **Eclipse Version 3.1** installation packages of the required platform (Windows/HP/Linux) from the location **http://www.eclipse.org/downloads/index.php.**
5. Unzip the package to a directory (for example /eclipse).
6. Download **C/C++ Development Tools Plug-in (CDT) Version 3.0.0** of the required platform(Windows/HP/Linux) from the location **http://download.eclipse.org/tools/cdt/releases/eclipse3.1/dist/3.0.0/**
7. Unzip the package to the same directory as in Step 2.
8. Unzip the SDE2 plug-in package, **sde2plugin_eclipse_2_3_Beta.tar.gz** into the same directory as given in Steps 2 and 4.
9. You should be able to see the directory structure as shown in Figure K-1.
10. Open a command prompt or unix/linux shell and set SDE2 supported configuration specific environment variables (run site specific batch files).
11. Change current working directory to \eclipse.
12. Invoke **eclipse** executable to launch eclipse. Eclipse IDE alongwith SDE2 plug-in would appear as shown in Figure K-2.

Figure K-1: Directory structure of Sde2_Eclipse Plugin

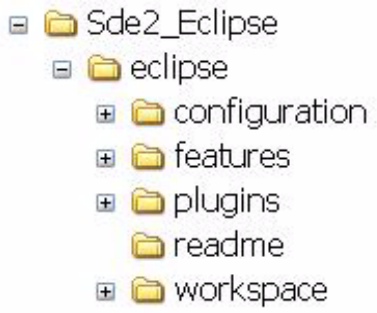
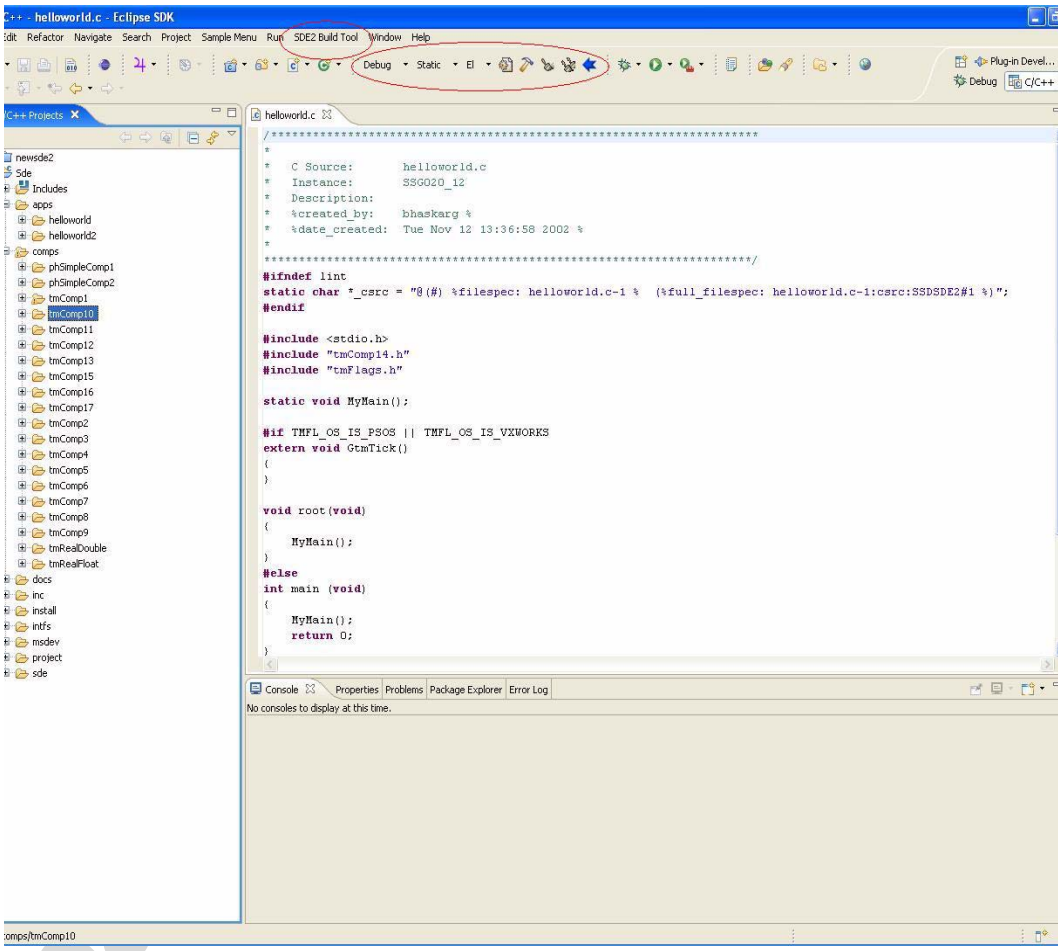


Figure K-2: Eclipse IDE after integration with SDE2 Plug-in



K.2.2 Adding project into Eclipse

1. Launch eclipse IDE as described above.
2. Add your components and applications as a C/C++ project as described below.
   a. Select **New** → **Project**. Expand **C** and select Standard Make C Project.
   b. In C/Make Project wizard, enter a Project Name. Uncheck “Use Default” , select the directory that contains the components and applications.
   c. Select the Environment tab, and add SDE2 environment variables one by one.
   d. Repeat the above steps for as many number of projects as required.
   e. Click on Finish.
3. Files and folders relevent to C/C++ projects are displayed as a tree structure.

Note: The user can run a batch file, which has SDE2 environment variables, for any known configuration before launching the Eclipse IDE. Only the requisite environment variables that he intends to change/modify, the user can add it in the environment tab. This avoids the user in entering each and every environment variable.

K.2.3 SDE2 menus

K.2.3.1 Context-sensitive menu

Context-sensitive-menu is provided on component/application makefiles. This pop-up menu appears when user right-clicks on any of the “makefile”. This pop-up menu contains the following SDE2 commands:

- **gmake**
- **gmake clean**
- **gmake clean_all**
- **build_exe**
- **auto_doc (devdoc and userdoc)**
- **Other options (Qmore, makelist, lint, QAC)**

Figure K-3: Pop up menu of SDE2 Plug-in

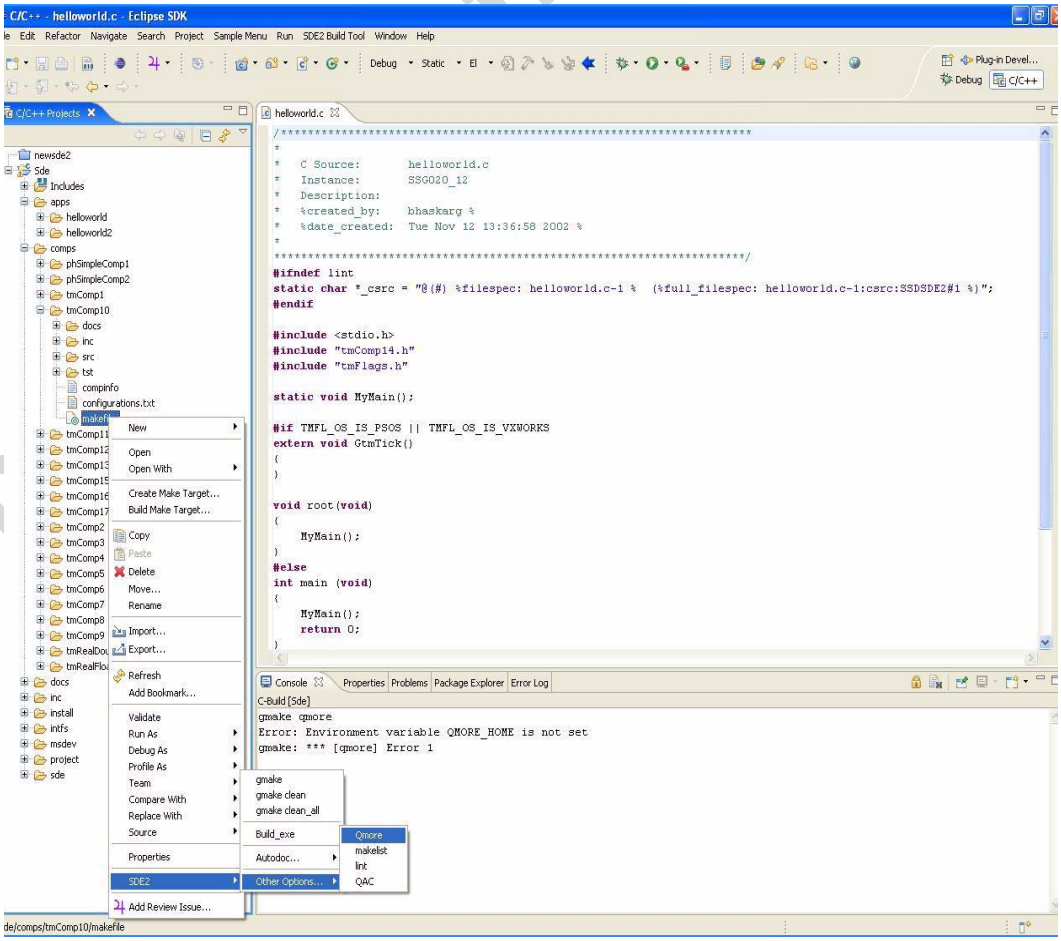
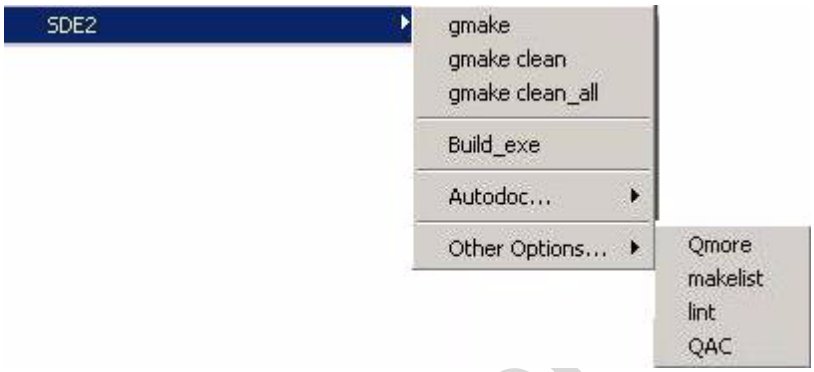


Figure K-4: SDE2 Pop-up menu options



K.2.3.2 SDE2 main menus

The SDE2 main menu integrated in Eclipse IDE as shown in the Figure K-5

Figure K-5: Main menu options of SDE2 plug-in

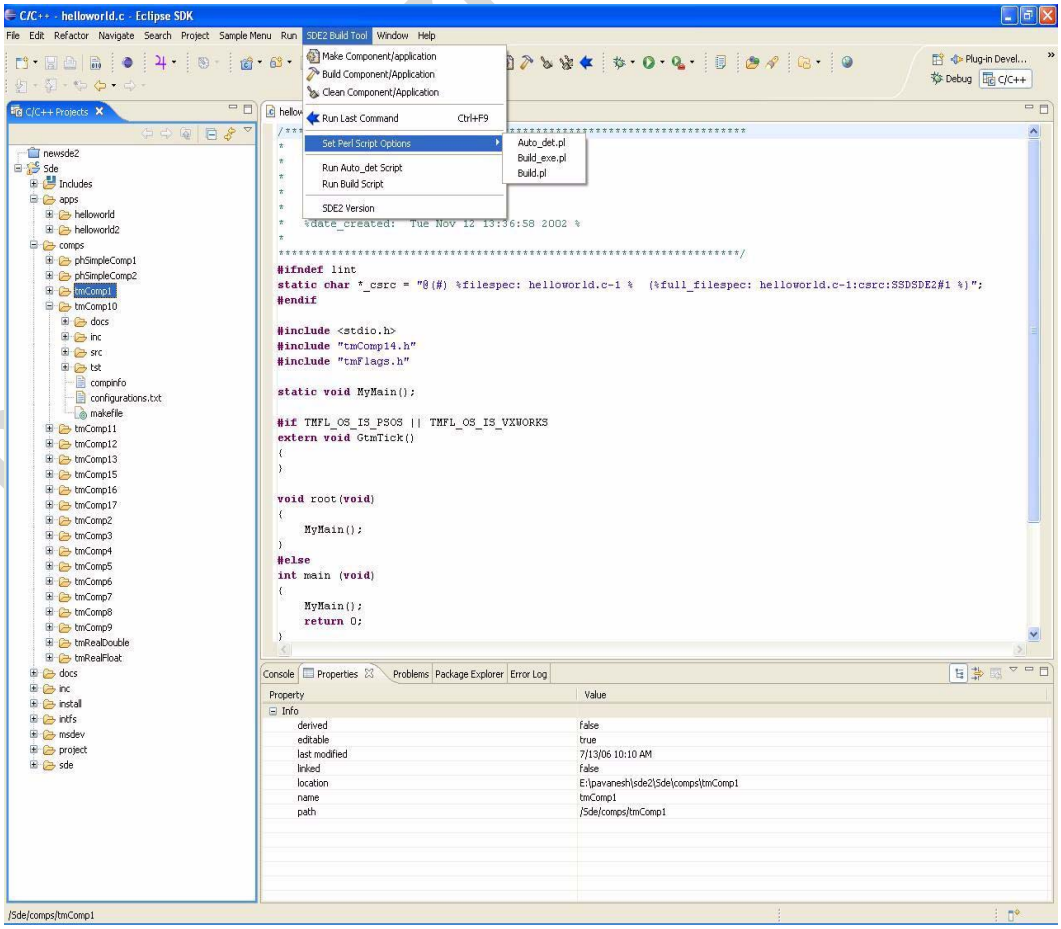
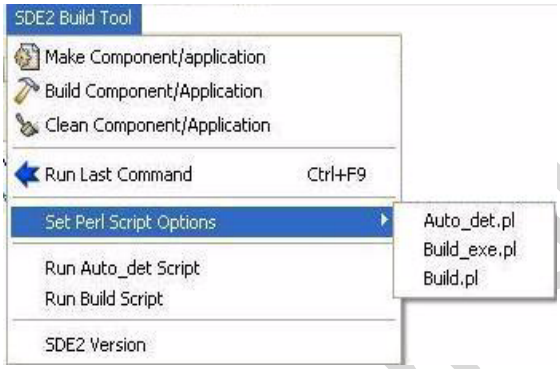


Figure K-6: SDE2 main menu options



Main menu “SDE2 Build Tool” can be used for the following SDE2 commands.

- Make a component/application
- Build a component/application.
- Clean a component/application.
- Run the last command executed.
- Selecting the options/parameters for all SDE2 perl scripts like **build_exe.pl, build.pl and auto_det.pl.**
- Run the script **build.pl.**
- Run the script **auto_det.pl.**
- Run the menu “**SDE2 Version**”, which prints the version of SDE2 being used onto the console.

User can also use short-cut keys to run the Perl scripts.

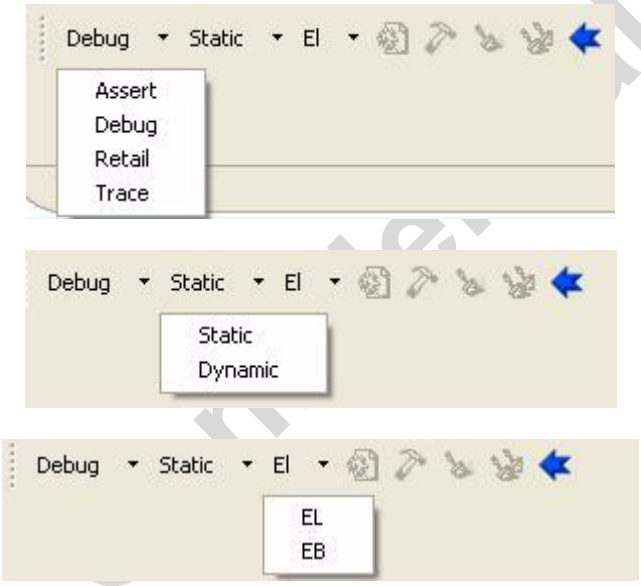
K.2.3.3 SDE2 toolbar options

Following toolbar options are provided.

1. Make a component/application
2. Build a component/application.
3. Clean a component/application.
4. Clean all components/application.
5. Run last command executed.

Figure K-7: SDE2 toolbar options

Options for changing Release, Linktype and Endianness dynamically.



K.2.3.4 SDE2 help menus

SDE2 getting started, user manual and frequently asked question and answers are provided in the help-contents within Eclipse IDE. SDE2 help is present at **Help** → **Help Contents.** Click on **SDE2 Help** to view getting started and user manuals. To view the FAQs click on the **FAQs of SDE2 Plugin.**

K.2.4 Change of environment variables

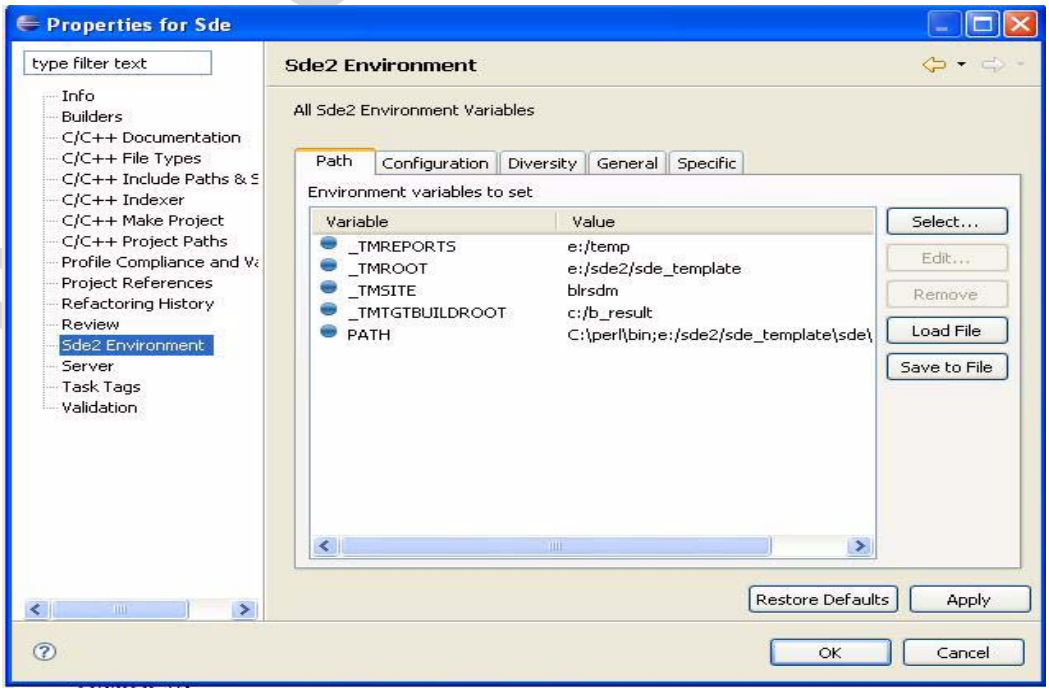
The environment variables required for SDE2 can be specified in the Properties of the Project -> SDE2 Environment property. To change the Environment, select the project and bring up the properties by right-clicking the project. Click on “SDE2 Environment” property. There are five tabs to change the Environment settings. They are: Path, Configuration, Diversity, Generic and Specific. Select one of the respective tabs to change the required Environment variables.

For eg:

All PATH related variables like _TMROOT, _TMTGTBUILDROOT, _TMPROJECT,

etc are listed in the PATH tab.

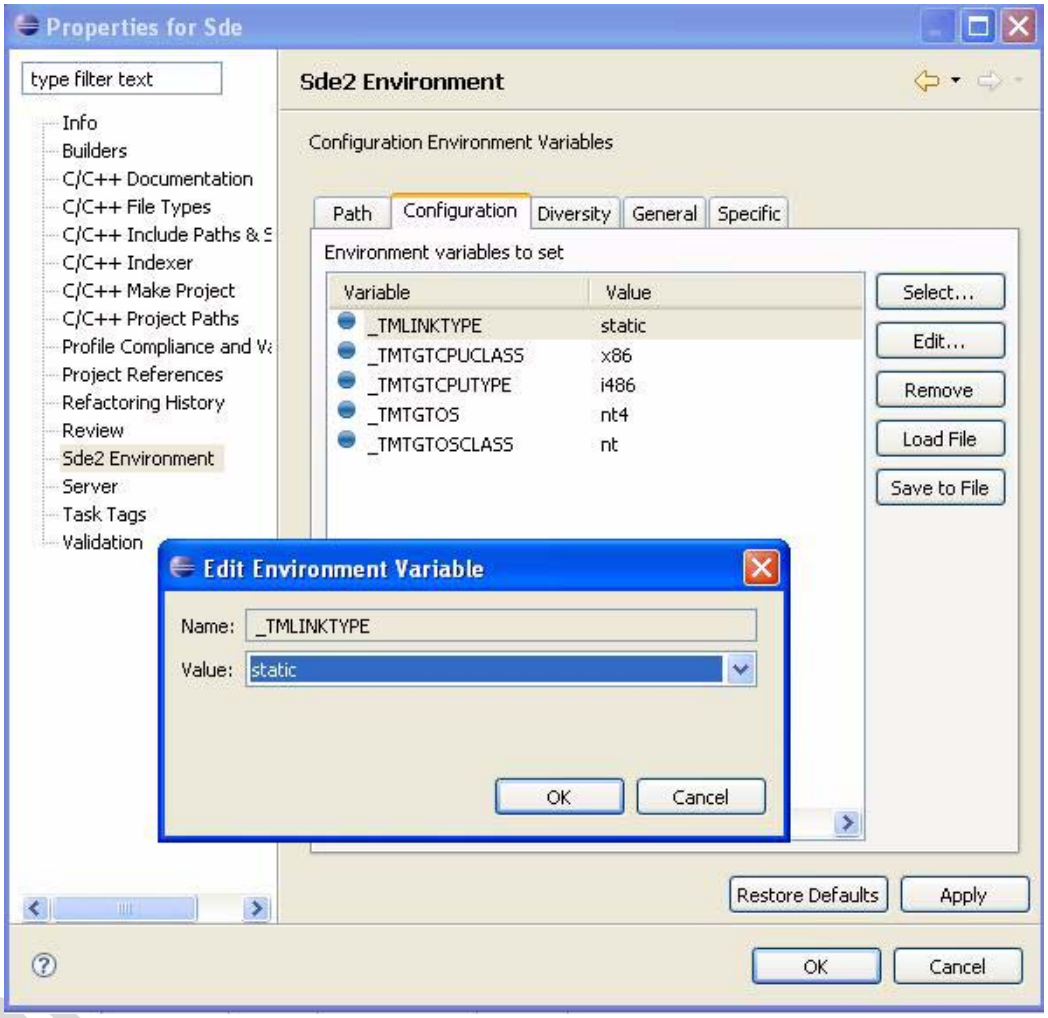
Figure K-8: Environment variable setting option



K.2.4.1 Changing an Environment variable

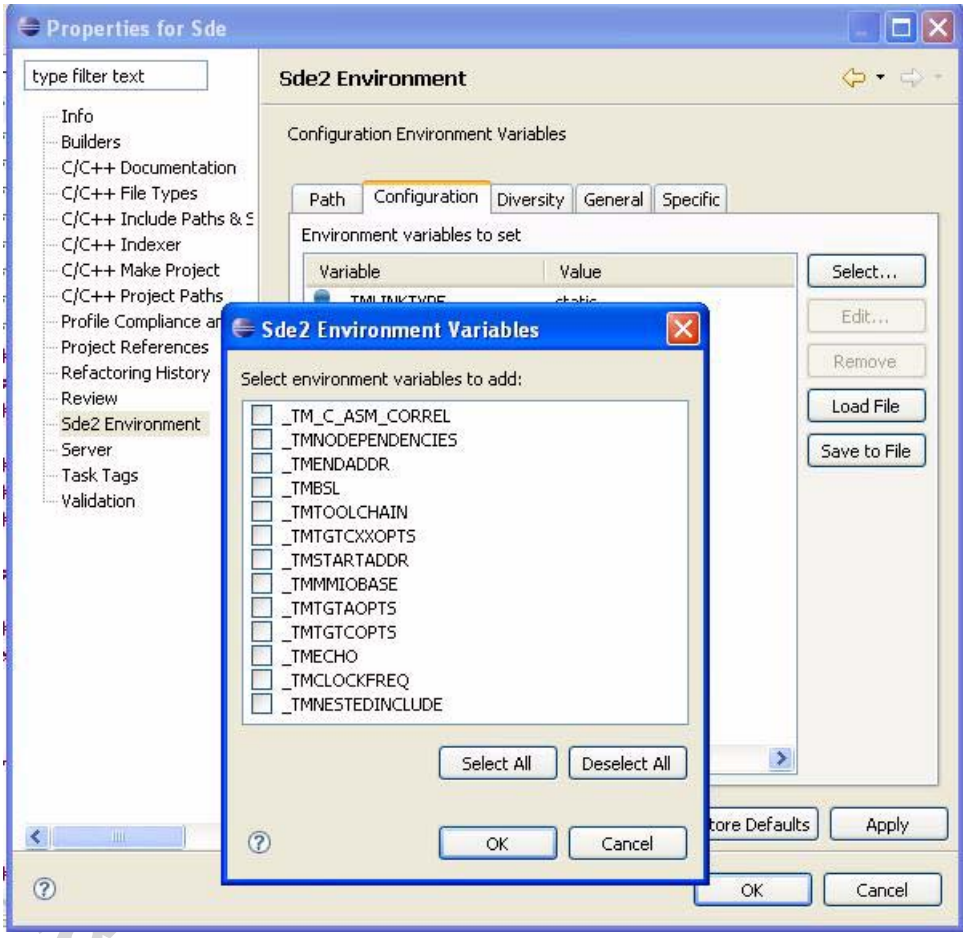
Select the environment variable that need to be changed and click edit or double click on the variable listed. This would popup a new window to change the variable.

Figure K-9: Changing an environment variable



K.2.4.2 Selecting an Environment variable not listed

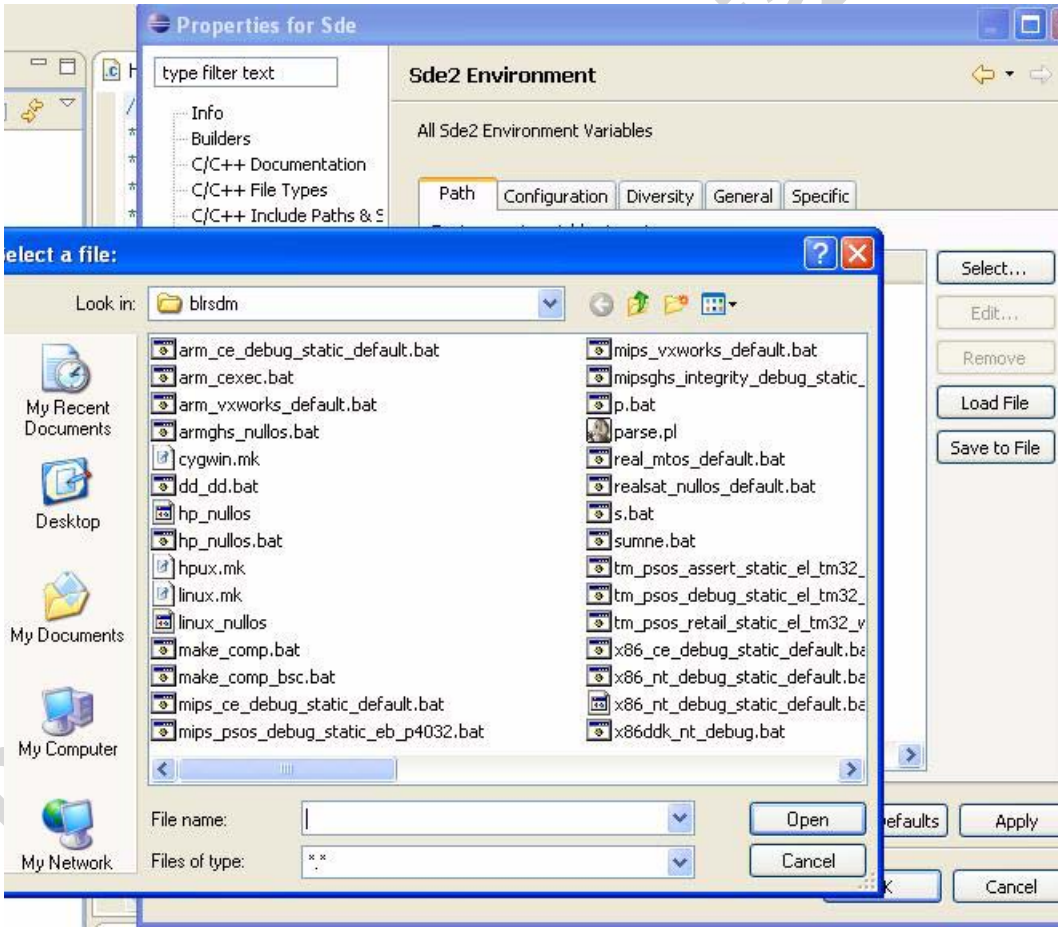
To select the SDE2 environment variables not listed in the any of the tabs click on the Select button and select the required variables.



K.2.4.3 Loading Environment from an SDE2 initialisation file

To load the environment variables from SDE2 initialisation files, click on the Load File Button and select the file to be loaded.

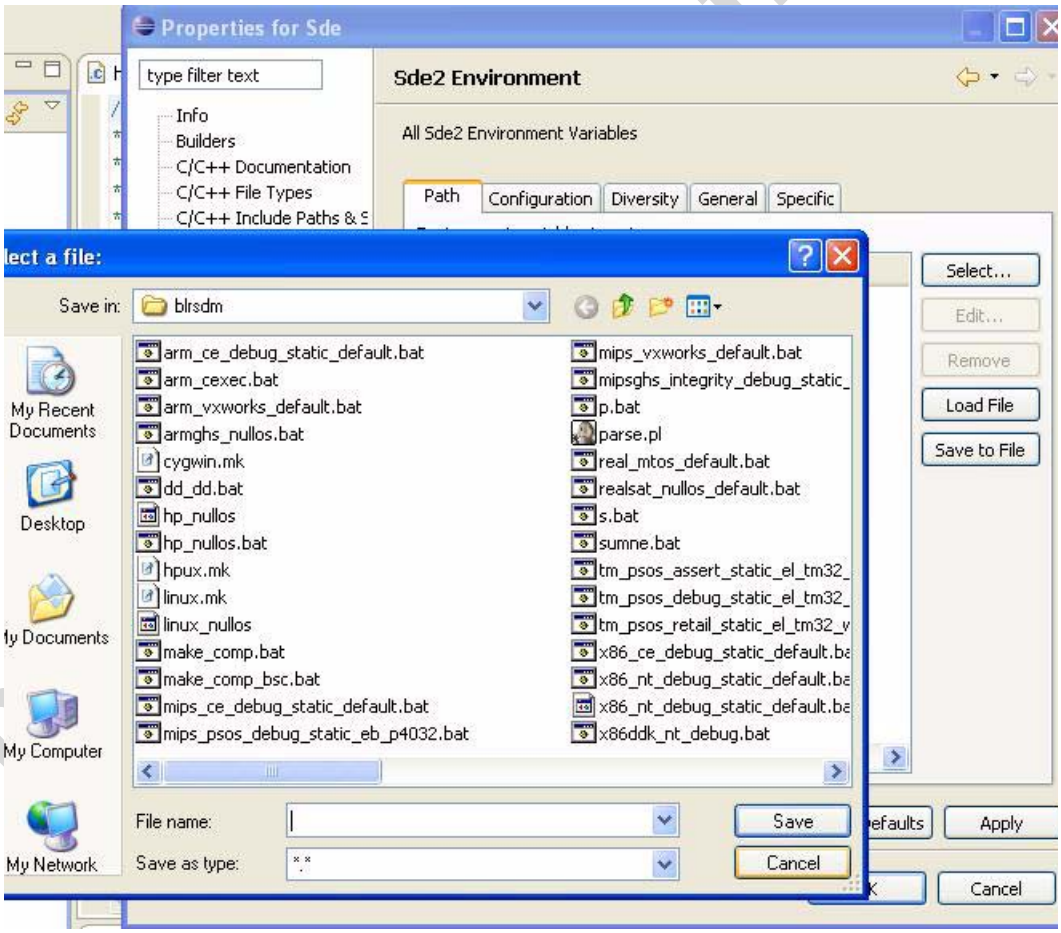
Figure K-10: Changing an environment variable



K.2.4.4 Saving environment variables set to a file

To save the environment variables set, click on the “Save to File” Button and select the name of the file to be saved.

Figure K-11: Saving environment variables set to a file



K.2.5 Display of build output

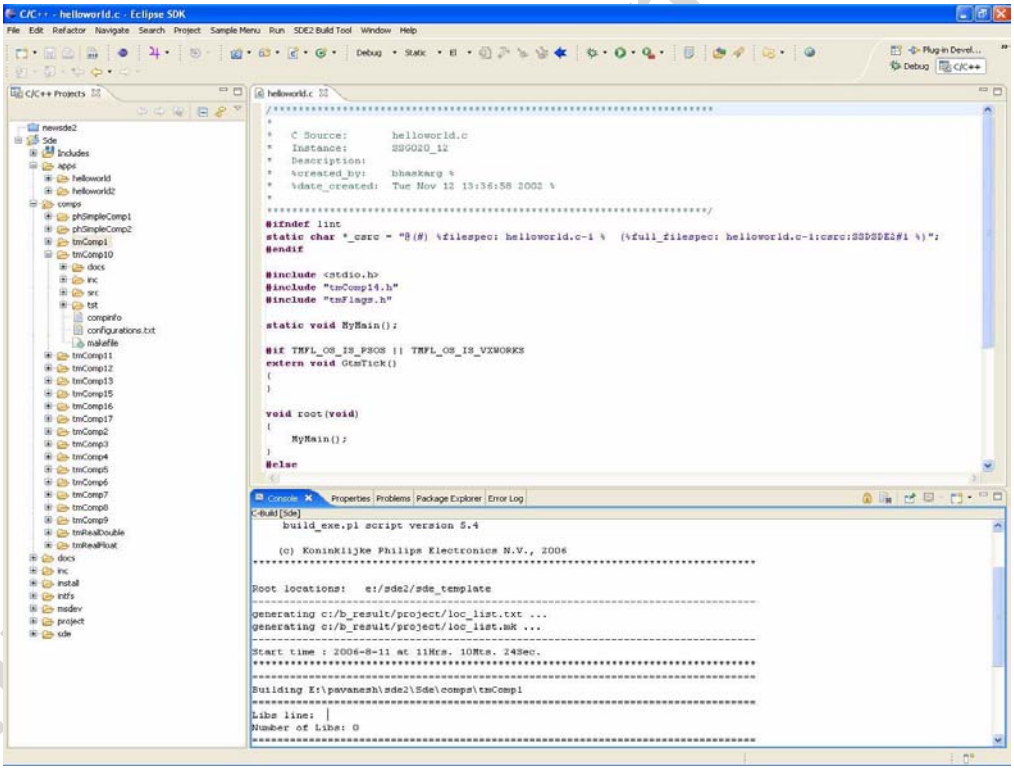
SDE2 plug-in makes use of the C/C++ Development Toolkit (CDT). CDT is a set of Eclipse plug-ins that provide C and C++ extensions to the Eclipse workbench. For more information on CDT, refer www.eclipse.org/cdt.

Eclipse and CDT provide many views which assist the user by providing structured information about edited/selected resource as below.

K.2.5.1 Console view

Console view displays the output of the SDE2 build tool. This view is enabled by choosing **Window** → **Show View** → **Other**. Select **Basic** → **Console** from the Show View dialog.

Figure K-12: Console view when a component is built using context-based drop-down menu on makefile.



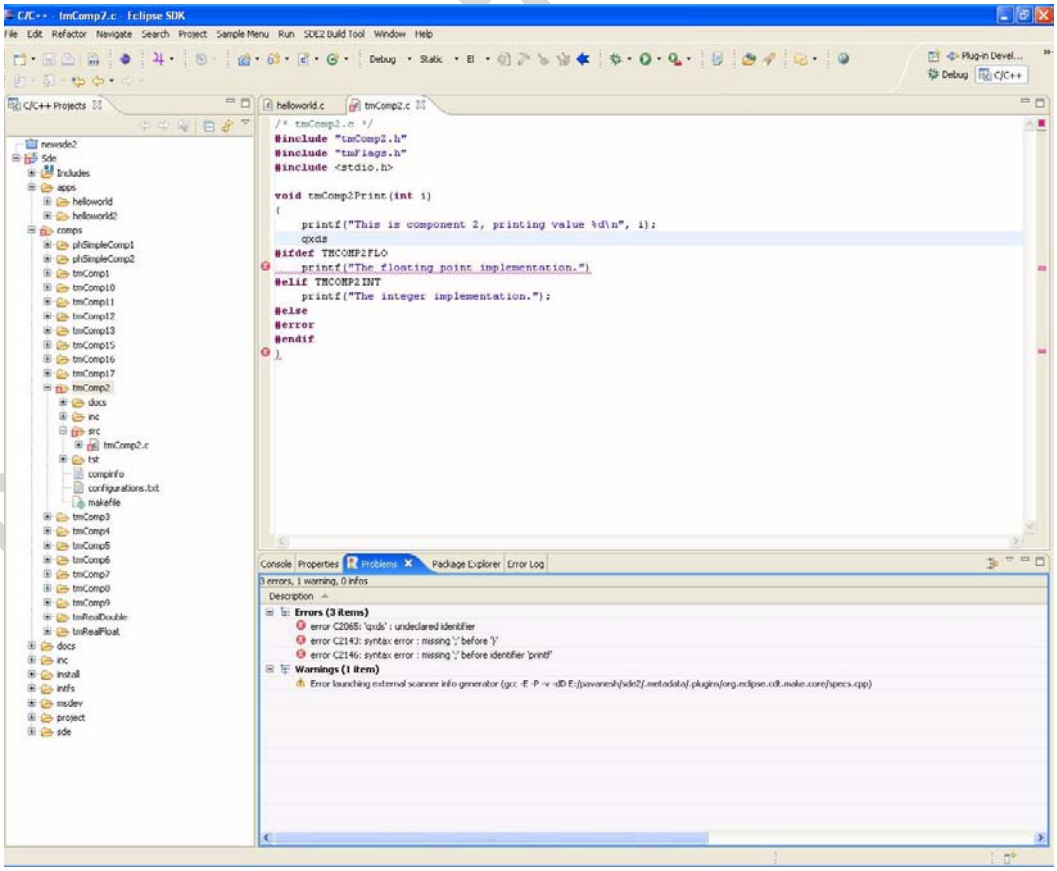
NOTE: Console window would appear in the foreground during the build to show the progress of the build being carried out.

K.2.5.2 Problems view

If you encounter any errors/warnings during a build, they will be displayed in the Problems view. This view is enabled by choosing **Window** → **Show View** → **Other**. Select **Basic** → **Problems** from the Show View dialog.

The user can view the description of error/warnings, line number and its location. User can navigate the sources with the indexed syntax errors obtained during SDE2 build, by double-clicking on the errors as shown in Figure K-13.

Figure K-13: Navigate the sources with the indexed syntax errors obtained during SDE2 build



K.2.5.3 Outline view

Displays the structure of the file currently open in an editor. This view is enabled by choosing **Window** → **Show View** → **Other**. Select **Basic** → **Outline** from the Show View dialog

Appendix L
Glossary

User Manual Version 3.8 **Sep 29, 2006**

Table L-1: Frequently used terms and abbreviations

Term	Description
API	Application Programmer's Interface
Application	Designed to function as a system. Complex, with handling of multiple cases and possible errors. Distributed with a collection of components, or as a product.
BSL	Board Support Library
BSP	Board Support Package
CCB	Change Control Board
CDT	C/C++ Development Tools
CM	Configuration Management
Component	A coherent and encapsulated piece of software whose interfaces are well defined and that was designed keeping reuse in mind, such that it can be deployed independently.
Configuration	Set of options determining what compiler with what settings is used for building products (libraries/executables) from source code.
Configuration class	The combination of CPU-class and operating system, used for building source files. Each configuration is part of a configuration class.
CMSynergy	Telelogic's Configuration Management Tool (earlier called Continuus)
CR	Change Request
Cygwin	UNIX environment for Windows
DLL	Dynamic Link Library
DVP	Digital Video Platform
Examples	These are like applications, but they are located with a library. They are designed to demonstrate usage of API. Examples tend to be simple. Generally distributed with a component library.
Executable	An application designed to be executed on a certain system. It may require other files. It is part of the final product consisting of applications, examples or tests. By executable we mean one of those three.
Generic	Describes SDE2's capability of being built for multiple configurations
GUI	Graphical User Interface
HVE-SWAB	High Volume Electronics SoftWare Architecture Board
Integrated	Used to describe SDE2's capability of including third-party tools at build time
IDE	Integrated Development Environment
JAR	Java Archive class file
OS	Operating System
OSAL	Operating System Abstraction Layer

Table L-1: Frequently used terms and abbreviations <Helv9R>(Cont'd.)

Term	Description
PDE	Plug-in Development Environment
PR	Problem Report
Production	Building a binary release using SDE2
Reusable	Refers to SDE2's capability of building reusable software
SDE2	SDE (Software Development Environment) consisting of a directory structure, makefiles and tools
Test	Application designed to verify functionality. May be large and tedious. Not generally distributed with the library.

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X