TC3xx BIFACES Template Projects

Configuration and Usage details to create own applications

2nd Generation AURIX™
32bit Microcontrollers

About this document

Scope and purpose

This document provides the details about ***BIFACES*** template projects, their configuration and usage of infrastructure ***API***s for basic application needs. This document provides the details about only selected Infrastructure ***API***s. You are suggested to refer the documentation of individual modules from ***iLLD*** documentation.

Intended audience

This template is intended for the users of ***BIFACES*** while using the ***Base Template Project*** or the projects which are derived from ***Base Template Project***/s.

Table of contents

Table of contents

1 Introduction to Base Template Projects

Base Template Projects provide a project framework/s to enable users of 2nd Generation AURIX™ microcontrollers. ***Base Template Project***s work together with ***BIFACES*** build environment. These projects come with defined folder structure and infrastructure drivers. These projects also contain the required configuration of build environment ***BIFACES***, for its basic usage.

BIFACES template projects are available for each microcontroller derivate (and silicon steps, which are in use) separately. These projects are packaged under workspace folder **Aurix2GWorkspace**. This is an Eclipse workspaces which could be launched with Eclipse CDT. Each template project is named as BaseFramework_<microcontroller derivate><step>. E.g. BaseFramework_TC39A, BaseFramework_TC39B etc.

1.1 Motivation

As you already know ***BIFACES*** is a generic build environment, which has used case with command line usage. Automatic launching of project, where the microcontroller derivate selection is possible, is not available. (refer to ***BIFACES*** User Manual). If you want to prepare a project from scratch, you have to prepare following basic setup:

- Configuration of ***BIFACES*** for used architecture, compiler and their tools
- Integrate the infrastructure drivers of ***iLLD*** for a given microcontroller
- Prepare the linker command file for a given microcontroller
- Configure clocks
- Implement the application entry points (coreX_main function for each TriCore™ CPU)
- Etc.

As ***Base Template Project***s prepare the above list for you upfront, they come very handy to save time and you could directly start on the application you are working on.

These template projects also come with standard folder structure, which makes it easy to handle files, integrate different modules that are coming from different sources.

1.2 Folder Structure

Folder structure of a typical project, which needs to be built with ***BIFACES***, is explained in "BIFACES User Manual" in detail with section "Project Structure". ***Base Template Project*** follow the recommendations made by this document.

1.3 Available Features

Base Template Project comes with set of infrastructure drivers, which enables you to start with development of application with 2nd Generation AURIX™ Microcontrollers.

1.3.1 Infrastructure Drivers

Following table provides the list of drivers that are available with ***Base Template Project***.

Table 1 Infrastructure Drivers

Driver	Package	Description
CPU	iLLD	***API***s to access CPU as a peripheral module
DMA	iLLD	***API***s to access DMA features
MTU	iLLD	***API***s to access the peripheral, Memory Test Unit

Table 1 Infrastructure Drivers (continued)

Driver	Package	Description
PMS	iLLD	***API***s to access the power management control with start-up software
PORT	iLLD	***API***s to access the ports and pins
SCU	iLLD	***API***s to access the system control unit for clock and watchdog functionalities
SRC	iLLD	***API***s to access the interrupt nodes
STM	iLLD	***API***s to access the system timer
SSW	SSW	Startup Software
REG	SFR	SFR Header files

There are other source files too which are normally the dependency files needed by above listed modules.

1.3.2 Default Hardware Configurations

Below table provides main parameters which are available as default configuration with template project.

Table 2 Default Hardware Configurations

Parameters	Values
Start Address	0xA0000000
Enable Lockstep Feature	All CPUs, which have such feature in the microcontroller
LBIST at Startup	Disabled
MONBIST	Disabled
Firmware Checker Software CHSW	Enabled
MBIST at Startup	Disabled
PMS EVR Init	Enabled
System PLL Frequency	Allowed maximum for the derivative. As an example: TC39 controllers are configured with 300MHz
Peripheral PLL Frequency	Allowed maximum for the derivative
Clock distribution to peripherals and buses	Allowed maximum values for the configured System Frequency and Peripheral PLL Frequency
Enabled CPUs	All available Tricore™ CPUs; HSM, SCR and GTM-MCS are not enabled by default
Caches	Both Data and Program Caches are enabled for all available Tricore™s

1.3.3 Default Build Environment Configurations

Below table provides main configurations, which are available as default configuration with template project.

Table 3 Default Build Environment Configurations

Parameter				Value
Architecture:	TriCore			Enabled
	Target:	Tc		Enabled
		Output:	elf	Enabled
		Output:	hex	Enabled
		Output:	srec	Disabled
		Output:	a	Disabled
	Toolchain	Dcc	All required tools	Enabled
		Ghs	All required tools	Enabled
		Gnuc	All required tools	Enabled
		Tasking	All required tools	Enabled

1.3.4 Used Compiler Tool-chain Options

Base Template Projects come with set of compiler toolchain options chosen based on commonly used use-cases. These are made available for each supported compiler toolchain. These are the set of options for which the template projects build successfully and also run on the Triboard™. However these options could be changed by you depending on your application need.

1.3.4.1 Used DCC Options

Following table provides the details about default options for DCC compiler:

Table 4 Dcc Compiler Options

Option	Rationale

-tTC162NF:simple	Generate code for TriCore™ v1.6.2 core (2nd Generation AURIX™ controllers);
	Single Hardware, Double Software Floating Point;

-O	Optimization for code and data
-XO	Extra optimization
-Xsection-split=1	Place each function in its own CODE section class and generates multiple instances of the split section for example, one .text section for each function
-Xkeep-assembly-file=2	Create and keep assembly file, the name of the .s file is the same as the name of the object file
-g3	Generate symbolic debugger information and do most optimizations
-Xinline=0	No automatic inline of functions
-Xdialect-c99	Comply to C standard list, select ISO C99
-Xabsolute18-data=0	Not to use (size=0) register absolute addressing mode for data (zbss, zdata)

Table 4 Dcc Compiler Options (continued)

Option	Rationale
-Xabsolute18-const=0	Not to use (size=0) register absolute addressing mode for constants (zconst, zconst)
-Xsmall-data=0	Not to use (size=0) register relative addressing mode for data (sbss, sdata)
-Xsmall-const=0	Not to use (size=0) register relative addressing mode for constants (sconst, sconst)
-ei5388,2273,5387	Suppress warning messages for 5388, 2273 and 5387

For DCC assembler same options as compiler options are used.
Following table provides the details about default options for DCC linker:

Table 5 DCC Linker Options

Option	Rationale
-tTC162NF:simple	Generate code for TriCore™ v1.6.2 core (2nd Generation AURIX™ controllers); Single Hardware, Double Software Floating Point;
-m6	Generate a more detailed link map of the input and output sections plus generate a link map with a cross reference table
-Xremove-unused-sections	Remove all unused sections

1.3.4.2 Used GHS Options

Following table provides the details about default options for GHS compiler:

Note: GHS compiler is only used with AURIX™ 2nd Generation Controllers

Table 6 GHS Compiler Options

Option	Rationale
-cpu=tc1v162	Generate code for TriCore™ v1.6.2 core (2nd Generation AURIX™ controllers);
-Ogeneral	Enables optimizations that improve both size and performance
-Onounroll	Disables loop unrolling if it has been enabled by an optimization strategy
-c99	Compliant with ISO/IEC 9899:1999 and does not allow any non-standard constructs
--no_commons	Allocates uninitialized global variables to a section and initializes them to zero at program startup
--gnu_asm	Enables GNU extended asm syntax support for most common use cases.

Table 6 GHS Compiler Options (continued)

Option	Rationale
--no_short_enum	Don't use smallest possible type for enum
-dwarf2	Enables the generation of DWARF debugging information in the object file
-minlib	Search only the target-specific run-time support library: libarch.a
-sda=none	No Small Data Area optimization (size=0)
-no_fused_madd	Prevents the compiler from generating multiply instructions where a floating-point multiply is combined with a subsequent addition or subtraction into a single operation
-no_float_associativity	Don't apply the associativity property to floating-point expressions
--no_vla	Disables support for variable length arrays (VLAs)
--diag_error 549,940,1056,1780,1999	Sets the specified compiler diagnostic messages to the level of error

Following table provides the details about default options for GHS assembler.

Table 7 GHS Assembler Options

Option	Rationale
-cpu=tc1v162	Generate code for TriCore™ v1.6.2 core (2nd Generation AURIX™ controllers);
-Ogeneral	Enables optimizations that improve both size and performance
-Onounroll	Disables loop unrolling if it has been enabled by an optimization strategy
-c99	Compliant with ISO/IEC 9899:1999 and does not allow any non-standard constructs
--no_commons	Allocates uninitialized global variables to a section and initializes them to zero at program startup
--gnu_asm	Enables GNU extended asm syntax support for most common use cases.
--no_short_enum	Don't use smallest possible type for enum

Following table provides the details about default options for GHS linker:

Table 8 GHS Linker Options

Option	Rationale
-cpu=tc1v162	Generate code for TriCore™ v1.6.2 core (2nd Generation AURIX™ controllers);

1.3.4.3 Used GNUC Options

Following table provides the details about default options for GNUC compiler:

Table 9 GNUC Compiler Options

Option	Rationale
-mtc162	Generate code for TriCore™ v1.6.2 core (2nd Generation AURIX™ controllers)
-g	Produce debugging information in the operating system's native format
-O2	Optimization level 2: balanced optimization for compilation time and the performance
-fno-common	No common block for uninitialized variables. Gives error if same variable is defined multiple times
-fshort-double	Use the same size for double as for float, this uses correct library for floating point arithmetic
-fstrict-volatile-bitfields	This option should be used if accesses to volatile bit-fields should use a single access of the width of the field's type
-ffunction-sections	Functions are placed in their exclusive sections. This option is required to remove unused functions during locate phase
-fdata-sections	Data are placed in their exclusive sections. This option is required to remove unused data during locate phase
-Wall	Enables set of important warnings which are defined by tool-chain

For GNUC assembler same options as compiler options are used.

Following table provides the details about default options for GNUC linker:

Table 10 GNUC Linker options

Option	Rationale
-mtc162	Generate code for TriCore™ v1.6.2 core (2nd Generation AURIX™ controllers)
--gc-sections	Enable garbage collection, which removes of unused input sections
-fshort-double	Use the same size for double as for float, this uses correct library for floating point arithmetic
-nostartfiles	Startup files shall not be used from compilers' standard library
-n	Turn off page alignment of sections

1.3.4.4 Used Tasking Options

Following table provides the details about default options for Tasking compiler:

Table 11 Tasking Compiler Options

Option	Rationale
--core=tc1.6.2	Generate code for TriCore™ v1.6.2 core (2nd Generation AURIX™ controllers)
--iso=99	Comply to C standard list, select ISO C99
-O2	Optimization level 2: balanced optimization for compilation time and the performance. (default level)
-g	Enable GCC extensions
--misrac-version=2012	C Code Checking: MISRA C version 2012

Following table provides the details about default options for Tasking assembler:

Table 12 Tasking Asm Options

Option	Rationale
--list-format=L1	List all list options
--optimize=gs	Allow generic instructions and Optimize instruction size

Following table provides the details about default options for Tasking linker:

Table 13 Tasking Linker Options

Option	Rationale
-OtcxyL	t=Compress copy table; c= Delete unreferenced sections; x=Delete duplicate code; L= Disable first fit decreasing
--core=mpe:vtc	Linker command file defines the vtc as virtual core comprising of all cores

2 Building Base Template Project

To build ***BIFACESBase Template Project***, you need to start ***BIFACES*** environment (Please refer to "BIFACES User Manual", section: "Working with BIFACES"). This section provides quick steps to start with a ***BIFACESBase Template Project***.

2.1 Prerequisites

Following are the prerequisites to use the ***BIFACESBase Template Project***s

- ***BIFACES*** installed locally on the machine. (Refer to section Installation in "BIFACES User Manual").
- ***BIFACESBase Template Project***s are extracted and made available as local folder path. It is recommended (not mandatory) to use the ***BIFACES*** workspace for Eclipse usage. (Refer to section "BIFACES Workspaces").
- Build environment launched (Refer to section "Launching Command-line/Eclipse Build Environment" in ***BIFACES*** User Manual).
- With Eclipse usage, ***Base Template Project***s are already launched. If not, please launch (Refer to section "BIFACES User Manual"- "Launching Project") or import (Refer to section "BIFACES User Manual"- "Importing Project") the projects explicitly.
- With command-line usage, project root is the current working directory (CWD).

2.2 Compiler Toolchain Setup

BIFACES needs to know which compiler toolchain is needed for your application and it also needs to know the path, where it is installed.

Following are the configurations that are needed to be considered.

2.2.1 Primary Toolchain

Primary Toolchain tells the ***BIFACES*** to use a specific ***Toolchain*** configuration, among multiple ***Toolchain*** configurations, as default ***Toolchain***. ***Base Template Project***s are pre-configured to support one of the “Gnuc”, “Tasking” or “Dcc” ***Toolchain***s for TriCore™ ***Architecture*** , and by default this value is “Gnuc”. You just have to tell which one is to be used by your application.

Primary Toolchain configuration is done at Config.xml file under the folder 1_ToolEnv/0_Build/1_Config.

Following are the steps to configure this parameter:

- Open configuration file Config.xml under folder 1_ToolEnv/0_Build/1_Config
- Look for the "architecture" configuration elements and select the one with attribute as: name="<Architecture Name>"
- In selected architecture element, modify the attribute "primaryToolchain" to the required ***Toolchain*** name. For ***Base Template Project***s, the allowed values are “Gnuc”, “Tasking” or “Dcc”.
- Save the file

Below is an example with Tasking ***Toolchain*** as ***Primary Toolchain*** for TriCore™ architecture.

```
<!-- Architecture configurations -->
<architecture name="Tricore" primaryToolchain= "Tasking">
       :
       :
  <!-- Toolchain configurations -->
  <toolchain name="Gnuc"
             enable="true"
             path="$(B_GNUC_TRICORE_CC_OPTIONS)"
             configFolder= "$(B_CONFIG_FILES_FOLDER)/Config_Tricore"
             configFiles= "1_ToolEnv/0_Build/1_Config/
Config.mk,Config_Gnuc.mk">
       :
  </toolchain>
       :
</architecture>
```

2.2.2 Toolchain Path

BIFACES knows the path to ***Toolchain*** through this configuration. This configuration is available with make variable. With ***Base Template Project***s this configuration is made available with following make expression:

B_<Toolchain>_<Architecture>_PATH: <Installation path>

Installation path = Path up to (and excluding) folder “bin”, where the required tools are stored.

For GNUC, the make variable is “**B_GNUC_TRICORE_PATH**” that is made available under file: "**1_ToolEnv/0_Build/1_Config/Config_Tricore_Gnuc/Config_Gnuc.mk**".

For Tasking, the make variable is “**B_TASKING_TRICORE_PATH**” that is made available under file: "**1_ToolEnv/0_Build/1_Config/Config_Tricore_Tasking/Config_Tasking.mk**".

For DCC, the make variable is “**B_DCC_TRICORE_PATH**” that is made available under file: "**1_ToolEnv/0_Build/1_Config/Config_Tricore_Dcc/Config_Dcc.mk**".

For example the path for GNUC is set as below:

```
#File: 1_ToolEnv/0_Build/1_Config/Config_Tricore_Gnuc/Config_Gnuc.mk

B_GNUC_TRICORE_PATH:= C:\Tools\Compilers\HighTec\V46X\toolchains\tricore
\v4.6.5.0
```

The above format is in general true for all the ***Toolchain***s used in ***BIFACES*** builds.

With Template Projects,

- Open the file which corresponds to configured ***Toolchain*** with your primaryToolchain.
- Modify the path and save.

*Tip: If you are developing application example for multiple compilers, you may keep all **Toolchain** paths configured. Then it is easy to switch between **Toolchain**s just by changing the primaryToolchain.*

2.3 Start Build

This section provides the details about how to start build, which is very obvious for many experienced users. To start the build activity, you need to invoke build target "**all**".

2.3.1 Usage with Command Line Environment

To start the build activity with command-line interface use the command: **make all**

Note: Refer to section “Invoking the Build Targets” in "BIFACES User Manual" to know about available make targets.

2.3.2 Usage with Eclipse Environment

Eclipse users could start the build activity in different ways with the CDT work-bench. One of such ways is explained at section “Invoking the Build Targets” in BIFACES User Manual.

There is another easy and quick option available as below:

- At the project explorer window, right-click on the project root folder. The menu opens up as shown below:
- Click on “Build Project”
- This results in the build running and finally the output is generated

2.4 Build Output

Build activity generates the make files first and then invokes the generated make files to generate the executable targets.

2.4.1 Generated Make Files

Make files are generated incrementally. That means, they are generated, when there is change in the folder structure or when there is change in project configuration.

Generated make files are available under 1_ToolEnv/0_Build/9_Make. These files are organized under the folder structure in the same way as source folders. Each SubDirectory.mk corresponding to a parent folder would include the SubDirectory.mk files that are corresponding to its child directories.

2.4.2 Executable files and Intermediate files

Build process generates the executables and build intermediate files under 2_Out/<Architecture>_<Primary Toolchain>. For an example with **Gnuc** as primary toolchain, output is generated at **2_Out/Tricore_Gnuc**.

The executable file BaseFramework_<derivate>.elf and map file BaseFramework_<derivate>.map are available is generated directly under 2_Out/Tricore_Gnuc.

Note: *The binary files such as BaseFramework_<derivate>.hex and BaseFramework_<derivate>.srec are configured as output in Base Template Projects but are not enabled by default. You need to enable these outputs if needed.*

Intermediate files such as generated asm files, dependency files and object files are generated to the hierarchical folder structure, which is same as that of source folders.

3 Creating My Application

Once you have your ***Base Template Project*** buildable, you could now start working on your application. This section provides descriptions and recommendations to create your application.

3.1 Prerequisites

You must have the ***Base Template Project*** configured to build and generate the executable for the required compiler ***Toolchain***. You must also check that the ***Base Template Project***s actually runs and stays in infinite loop of **coreX_main** functions. Please refer to section ***Building Base Template Project***

3.2 Copy Template Project as My Application

To create your own application, you are recommended to copy the working ***Base Template Project*** as your application. By doing this you are preserving a working ***Base Template Project*** for your next application. This can be done with normal copy of folder tree when you are not using eclipse or this can also be done with eclipse project explorer with its copy - paste feature.

Note: *When you are using eclipse as your editor, use the eclipse project explorer to copy. This saves your time to import the project to eclipse workspace*

3.2.1 Usual Copy Operation

Following are the steps to copy project, when you are not using eclipse:

1. At your workspace, copy the folder tree of ***Base Template Project*** as new folder tree. (e.g. BaseFramework_TC39B → MyApplication_TC39B)

   Tip: *Preserve the derivate name as postfix to keep track of compatible hardware*

2. **Clean** the project and **build** to be sure to have the new project is building for you

3.2.2 Working with Eclipse Environment

Following are the steps to build project when you are working with eclipse as an editor for ***BIFACES*** projects:

Copy the working ***Base Template Project***

- Under eclipse Project explorer: Copy the ***Base Template Project*** (which is buildable and also runs on the board, e.g. BaseFramework_TC39B) with Ctrl+c keys (or right-mouse-click on project root folder and click **Copy**).
- Paste Ctrl+v (or right-mouse-click at the end of folders can click **Paste**). You need to provide a new name in the window, which comes up. Give any new name which meaningful (e.g. MyApplication_TC39B). This is new project which has all the required settings which you have already done in template project.

3.3 Examples for Template Projects

Before creating any application, have a look into folder **<workspace you recently installed>/_Examples**. You see sources which you might already find useful for your application. Normally these examples are subset of required files, which could easily be merged with ***Base Template Project***s. These folders are named to indicate which feature they show. Also these examples contain the user guide, which also explain the objective and how to use instructions.

3.4 Your Application Files Folder

BIFACES would build your source files irrespective of its location within the source folders. However it is recommended to make a specific container for your application files because of following reasons.

- Identify your source files easily
- It is easy to port your application from one hardware to another, where you need to copy only the files that are created by you
- It is also easy to upgrade with latest version ***Base Template Project***s

Below figure shows the relative location as folder MyApplication. The name "MyApplication" is just an example. Please use the meaningful name for your application (eg. Pwm_Demo, BlinkeyLed_Demo etc.)

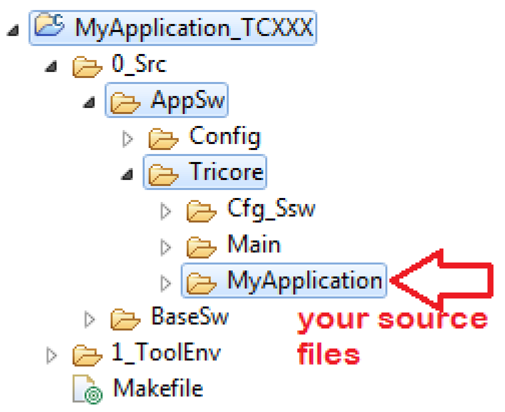

Figure 1 My Source

Tip: Organize the files with sub folders. ***BIFACES*** *creates make files automatically any way!*

3.5 Change System Frequency

System frequency is configured at the startup software based on the #define constants which are configured by user. Please refer to ***iLLD*** driver code, which is available in folder 0_Src\BaseSw\Infra\Ssw\<derivate> and 0_Src\BaseSw\iLLD\<derivate>\Tricore\Scu, to know how this is implemented. Scu drivers from ***iLLD*** provide set of pre-defined configurations which configure following parameters:

- System and Peripheral PLL frequencies

- Clock distribution with max allowed divider values for selected PLL frequencies
- Minimum possible Flash Wait State for the selected highest CPU frequency

Below is the table, which provide #define constant names for frequency configurations

Table 14 Configuration constants for clock configurations

#define Constant	Purpose	Values
IFX_CFG_SCU_XTAL_FREQUENCY	Crystal frequency	Allowed values: 16/20/40MHz: Where 20MHz= 20000000U
IFX_CFG_SCU_PLL_FREQUENCY	System Frequency	Allowed values: Frequency: 80/133/160/200/300MHz: Where 300MHz= 300000000U
IFX_CFG_SCU_PLL1_FREQUENCY	Peripheral 1 PLL Frequency	Allowed values: 320MHz= 320000000U
IFX_CFG_SCU_PLL2_FREQUENCY	Peripheral 2 PLL Frequency	Allowed values: 200MHz= 200000000U

Following is the example for crystal and system frequency configurations :

```
#define IFX_CFG_SCU_XTAL_FREQUENCY     (20000000)  /**< 16000000U/20000000U/
40000000U */
#define IFX_CFG_SCU_PLL_FREQUENCY      (300000000) /**< 80000000U/133000000U/
160000000U/200000000U/300000000U*/
```

3.6 Enable/Disable CPUs at Startup

Enable or Disable of CPUs are done at the startup software based on the following #define constants. Please refer to ***iLLD*** startup code, which is available in folder 0_Src/BaseSw/Infra/Ssw/<derivate>/Tricore, to know how this is implemented. When you disable a CPU with this configuration, user startup software would keep the CPU in halt mode. In case of CPU0 is disabled with this configuration, it goes to halt mode at the end of user startup software after starting other enabled CPUs.

Table 15 Configuration constants for CPU enable / disable

#define Constant	Purpose	Value
IFX_CFG_SSW_ENABLE_TRICORE**x**	Enable/ disable TriCore™ CPU **x** (where x=0-6) during user startup software	0: Disables 1: Enables

You need to define these #define constants explicitly only when you want to change the default value. If these are not defined explicitly, by default, all the TriCore™ CPUs are enabled by startup software. You need to define these constants at file **"0_Src/AppSw/CpuGeneric/Config/Ifx_Cfg.h"**.

When a CPU is not enabled during startup software, obviously, this would not participate in CPU synchronization event. CPU synchronization is done with Cpu-Emit and Cpu-Wait response calls. To make Cpu-Wait bypassed for a disabled CPU, following #define constant need to be configured:

Table 16 CPUs' emit event constant

#define Constant	Purpose	Value
IFXCPU_CFG_ALLCORE_DONE	**IfxCpu_emitEvent** API, which is called by each CPU during synchronization. This API updates a variable parameter corresponding to bit position represented by CPU ID. This constant value is the final value once all CPUs are synchronized. The variable is polled against this value by the API **IfxCpu_waitEvent**	Bit 0 = 1 : CPU0 is at synch point Bit 1 = 1 : CPU1 is at synch point Bit 2 = 1 : CPU2 is at synch point Bit 3 = 1 : CPU3 is at synch point Bit 4 = 1 : CPU4 is at synch point Bit 6 = 1 : CPU5 is at synch point

You need to define this #define constant explicitly only when you want to change the default value. If this is not defined explicitly, by default, a constant is defined that expect all TriCore™ CPUs to participate in CPU synchronization event. You need to define the above constant at file **"0_Src/AppSw/CpuGeneric/Config/Ifx_Cfg.h"**

Note: *In the worst case, if the above constant is not defined with appropriate value, the wait for CPU synchronization event will timeout and goes further. No error notification done in this case. However, is such a case, CPUs are not synchronized.*

Following is an example to disable CPU1 (as described earlier, by default all the CPUs are enabled). You need to modify the file Ifx_Cfg.h as shown below.

```
//File: 0_Src/AppSw/CpuGeneric/Config/Ifx_Cfg.h

#define IFX_CFG_SSW_ENABLE_TRICORE0  1
#define IFX_CFG_SSW_ENABLE_TRICORE1  0
#define IFX_CFG_SSW_ENABLE_TRICORE2  1
#define IFX_CFG_SSW_ENABLE_TRICORE3  1
#define IFX_CFG_SSW_ENABLE_TRICORE4  1
#define IFX_CFG_SSW_ENABLE_TRICORE5  1


//Constant for CPU Sync:
#define IFXCPU_CFG_ALLCORE_DONE                \
        (IFX_CFG_SSW_ENABLE_TRICORE0 << 0U) | \
        (IFX_CFG_SSW_ENABLE_TRICORE1 << 1U) | \
        (IFX_CFG_SSW_ENABLE_TRICORE2 << 2U) | \
        (IFX_CFG_SSW_ENABLE_TRICORE3 << 3U) | \
        (IFX_CFG_SSW_ENABLE_TRICORE4 << 4U) | \
        (IFX_CFG_SSW_ENABLE_TRICORE5 << 6U)
```

3.7 Enable/Disable Caches at Startup

CPU data cache and program caches are independently enabled/ disabled. This is done at the startup software, based on the following #define constants. Please refer to ***iLLD*** driver code, which is available in folder "**0_Src/BaseSw/Infra/Ssw/<derivate>/Tricore**" to know how this is done.

You need to define these #define constants explicitly only when you want to change the default value. If these are not defined explicitly, by default, all the caches with TriCore™ CPUs are enabled by startup software.

Table 17 Configuration constants for CPU cache enable / disable

#define Constant	Purpose	Value
IFX_CFG_SSW_ENABLE_TRICORE**x**_PCACHE	Enable/ disable program cache for CPU **x** (where x=0-6) during user startup software	0: Disables 1: Enables
IFX_CFG_SSW_ENABLE_TRICORE**x**_DCACHE	Enable/ disable data cache for CPU **x** (where x=0-6) during user user startup software	0: Disables 1: Enables

Above macros are imported by user startup software through file: "**0_Src/AppSw/CpuGeneric/Config/Ifx_Cfg.h**". You have to update this file to configure CPU enable or disable with user startup software.

Following is an example to disable CPU1 program cache and data cache (By default all the CPU caches are enabled). You need to modify the file as shown below.

```
//File: 0_Src/AppSw/CpuGeneric/Config/Ifx_Cfg.h

#define IFX_CFG_SSW_ENABLE_TRICORE0_PCACHE  1
#define IFX_CFG_SSW_ENABLE_TRICORE0_DCACHE  1
#define IFX_CFG_SSW_ENABLE_TRICORE1_PCACHE  0
#define IFX_CFG_SSW_ENABLE_TRICORE1_DCACHE  0
#define IFX_CFG_SSW_ENABLE_TRICORE2_PCACHE  1
#define IFX_CFG_SSW_ENABLE_TRICORE2_DCACHE  1
#define IFX_CFG_SSW_ENABLE_TRICORE3_PCACHE  1
#define IFX_CFG_SSW_ENABLE_TRICORE3_DCACHE  1
#define IFX_CFG_SSW_ENABLE_TRICORE4_PCACHE  1
#define IFX_CFG_SSW_ENABLE_TRICORE4_DCACHE  1
#define IFX_CFG_SSW_ENABLE_TRICORE5_PCACHE  1
#define IFX_CFG_SSW_ENABLE_TRICORE5_DCACHE  1
```

3.8 Using Interrupts

TriCore™ CPU supports both hardware managed interrupts and software managed interrupts. This mechanism is explained in figures below, where the interrupt mechanism of TriCore™ is shown in its simple form in a way that you understand it from software point view.

Interrupt vector base register, BIV, points to the vector table in each CPU. With TriCore™ architecture, you could setup vector tables in two ways that each CPU has its own vector table or each CPU access to one common vector table. Upon an interrupt trigger, CPU jumps to the correct vector in the vector table, based on interrupt priority and BIV configuration. You could place your vector table and/or interrupt service routines either in PFLASH or in PSPR. You need to configure the required Service Request Nodes, with the Priority Number and Type of Service.

TriCore™ implements interrupt mechanism in a nice and unique way that it can handle both the hardware managed or software managed interrupts easily, based on the vector table placement in the memory.

3.8.1 Hardware Managed Interrupts

With Hardware Managed Interrupts, the program jumps are made directly to the interrupt service routines (ISRs), which are statically located. Which means, the vector BIV configuration, vector table and ISRs are statically decided during linking phase, based on the linker command file. Below picture simplifies this mechanism to show how the hardware managed interrupt mechanism with Base Template Projects is implemented. Base template projects configure each interrupt vector size to be 32 byte, which in turn restricts the vector table location to be aligned to 8K address in the memory.

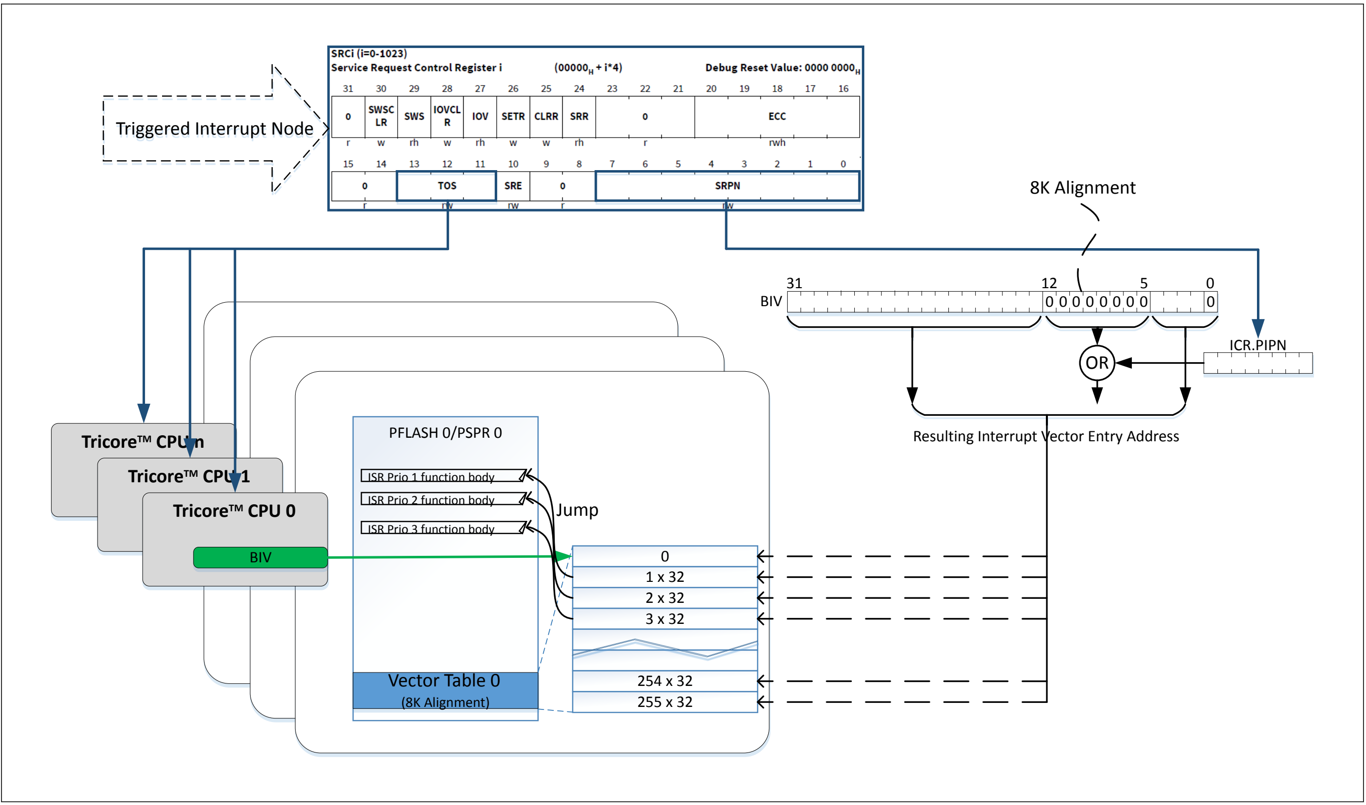

Figure 2 Hardware Managed Interrupts

For AURIX™ 2nd Generation Microcontrollers, the ***Base Template Project***s implement separate vector tables for each TriCore™ CPU. This allows us to locate the vector tables in memory, which have quick access possibility with associated CPU. For AURIX™ 1st Generation Microcontrollers, the ***Base Template Project***s implement common vector table for all TriCore™ CPUs in flash memory, where the access time is same for all CPUs. A vector table contains up to 255 vectors of 32 byte size. Each vector table starts at an 8K aligned address.

Startup software configures Base Interrupt Vector register of each TriCore™ CPU to corresponding vector table address, where BIV.BIV= <vector table address> and BIV.VSS= 0 (corresponds to 32 byte size of vectors). Because of the correct memory alignment of vector table to 8K boundary, the bits BIV.BIV[5-12] are "0"s.

The macro **IFX_INTERRUPT** encapsulates the compiler specific implementations to define an ISR, define a vector entry to vector table with jump to ISR function and also define a linker section for the vector entry. This is explained in detail in later sections. ***Base Template Project***s, by default, locate such vector tables and the ISRs in to flash memory of each CPU for AURIX™ 2nd Generation Microcontrollers. You may also change the linker command file to locate these to PSPR of each CPU.

In your application, you configure the Service Request Node registers with required priority (SRC.SRPN) and assign the CPU (SRC.TOS), which services this interrupt request. Upon the occurrence of such interrupt, the interrupt controller will interrupt CPU by updating **ICR.CCPN= SRC.SRPN**. CPU switch to interrupt context (please refer to the HW Manual for details) and executes from vector address that is automatically calculated by formula: **vector address = BIV | (ICR.CCPN << 5)**. As detailed earlier, this vector entry contains the code to jump to the ISR function. The ISR function, upon execution, returns (rei instruction) to the interrupted function.

3.8.2 Software Managed Interrupts

With Software Managed Interrupts, the automatic program jumps are made to an interrupt handler, which later calls the ISR (as normal function call) based on the ICR.CCPN. Here the interrupt service routines (ISRs) are located anywhere in the memory as a normal functions and a software vector is implemented as an array of function pointers in RAM. In case of Software Managed Interrupts, software vector table needs not to be static, which could be modified during runtime. You need to install/reinstall the ISRs in to the software vector table, whenever needed by your application. Below picture simplifies this mechanism to show how the software managed interrupt mechanism with ***Base Template Project***s is implemented.

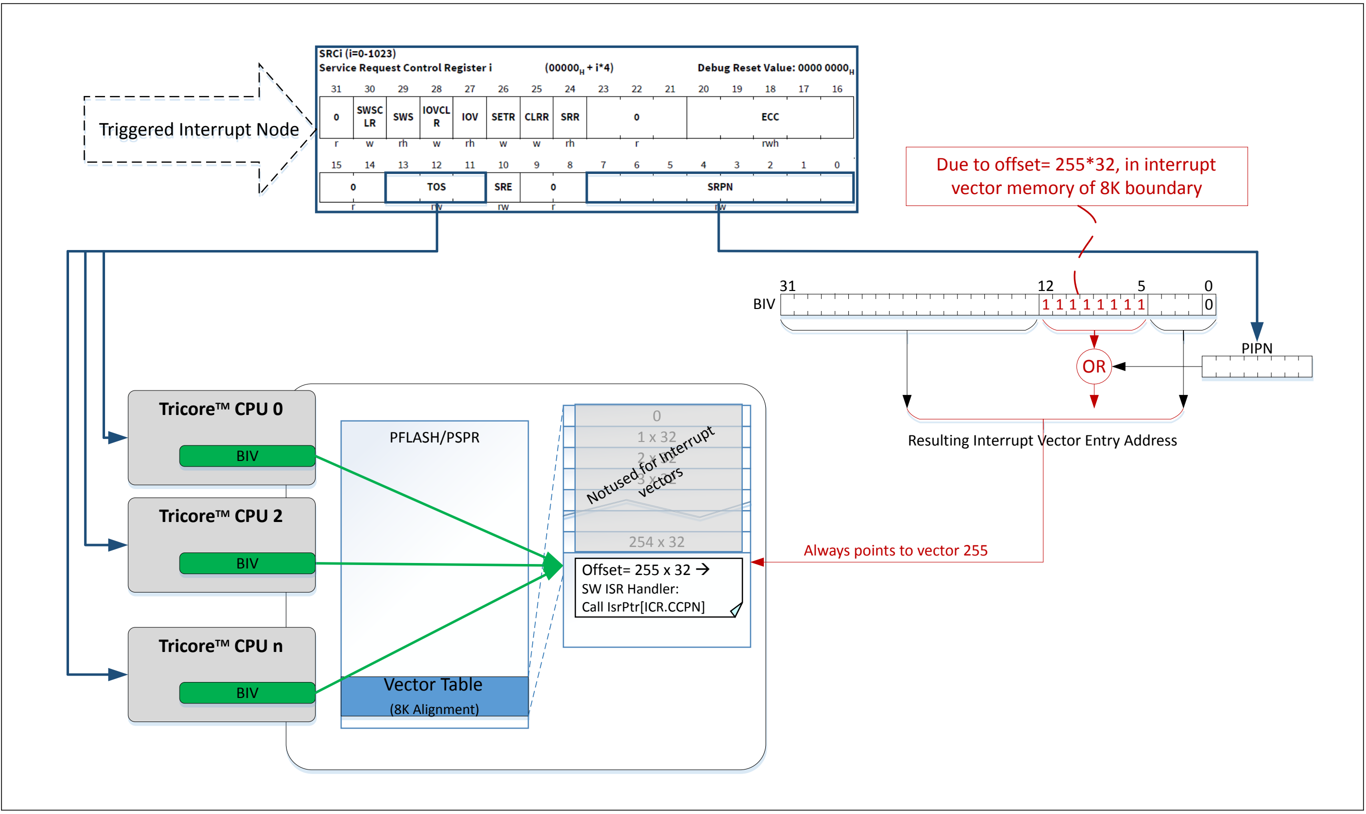

Figure 3 Software Managed Interrupts

Base Template Projects implement a common interrupt handler for all TriCore™ CPUs, which calls the ISR as function call. Interrupt handler is located at an address, which has an alignment to address 8K boundary with offset = 32*255. Because of such offset, address value has bit fields 5-12 as "1s". Startup software configures Base Interrupt Vector (BIV), to the address of such Interrupt handler.

In your application, you configure the Service Request Node registers with required priority (SRC.SRPN) and assign the CPU (SRC.TOS), which services this interrupt request. Upon the occurrence of such interrupt, the interrupt controller will interrupt CPU by updating **ICR.CCPN= SRC.SRPN**. CPU switch to interrupt context (please refer to the HW Manual for details) and executes from vector address that is automatically calculated by formula: **vector address = BIV | (ICR.CCPN << 5)**. All same as in hardware managed interrupts so far, except, in this case, the vector address which is (because of OR operations with "1"s at bits 5-12) pointing to the interrupt handler. CPU would jump to same vector entry for any interrupt priority or any CPU.

As explained earlier, interrupt handler calls now the ISR as function. The address function to call is fetched from the software vector, which is array of function pointers. You have installed (may also be updated during runtime) this function pointer exactly at the offset same as its ISR priority. Upon servicing such ISR, the function returns to interrupt handler. Interrupt handler then returns from interrupt to the interrupted function.

3.8.3 Advantage and Disadvantages

Following table lists advantages/ disadvantages between hardware managed and software managed interrupts:

Table 18 Hardware Managed and Software Managed Interrupts Differences

Objective	Hardware Managed	Software Managed
Latency	Low	High
Flexibility to modify ISRs during runtime	Not possible	Possible
Memory consumption	High	Low
Usage complexity	Low	High

3.8.4 Defining Interrupts in Application

This section provides the detailed steps about using the interrupt mechanism in your application with an example. This procedure is common for both hardware managed and software managed interrupts, unless specified explicitly.

Attention:* *In an application, you may either use hardware managed interrupts or software managed interrupts but not both.

Example shown below uses ***API***s from ***iLLD***, which implements an ISR for STM timer compare events. Such compare is configured for every 1s.

1. Define an ISR using macro IFX_INTERRUPT

```
 //file: <application file>

#define STM0_ISR_PRIORITY 10

/*Prototype from Ifx_Compilers.h: IFX_INTERRUPT(isr, vectabNum, prio) from
Intrinsics.h
* Isr: Function name of the interrupt service routine.
* VecttabNum: vector table number.
* Prio: interrupt priority
*/
IFX_INTERRUPT(Stm0_Isr, 0, STM0_ISR_PRIORITY)
{
    uint32 stmTicks;
    __enable();     //enables the high prio interrupts
    stmTicks = (uint32)(LED0_BLINK_INTERVAL_IN_SECONDS *
IfxStm_getFrequency(&MODULE_STM0));
    IfxStm_updateCompare(&MODULE_STM0, IfxStm_Comparator_0,
IfxStm_getCompare(&MODULE_STM0, IfxStm_Comparator_0) + stmTicks);
}
```

In case of hardware managed interrupts above definition:

- Defines an ISR function body with name **Stm0_Isr**
- Also defines a vector entry (with section statements as required for Base Template Projects) to jump to Stm0_Isr

In case of software managed interrupts above definition will simply define a function with name **Stm0_Isr**.

2. In the application files, configure the service request node (This is done by ***iLLD*** STM driver, please refer to the driver code)

```
/** Prototype from IfxSrc.h: IFX_INLINE void IfxSrc_init(volatile
Ifx_SRC_SRCR *src, IfxSrc_Tos typOfService, Ifx_Priority priority);
 * \brief Initializes the service request control register.
 * \param src pointer to the Service Request Control register which should
be initialised.
 * \param typOfService type of interrupt service provider.
 * \param priority Interrupt priority.
 * \return None
 */

IfxSrc_init (&MODULE_SRC.STM[0].SR[0], IfxSrc_Tos_cpu0, STM0_ISR_PRIORITY);
```

3. Enable the trigger

```
/** Prototype from IfxSrc.h: IFX_INLINE void IfxSrc_enable(volatile
Ifx_SRC_SRCR *src);
 * Enables a specific interrupt service request.
 * \param src pointer to the Service Request Control register for which the
interrupt has to be enabled.
 * \return None
 */
IfxSrc_enable (&MODULE_SRC.STM[0].SR[0]);
```

4. For software managed interrupts, configure the macro IFX_USE_SW_MANAGED_INT in the file Ifx_Cfg.h at location 0_Src\AppSw\Config\Common

```
// file:  0_Src\AppSw\Config\Common\Ifx_Cfg.h
#define IFX_USE_SW_MANAGED_INT
```

5. For software managed interrupts, install interrupt handler

```
/** Prototype from IfxCpu.h: IFX_EXTERN void
IfxCpu_Irq_installInterruptHandler(void *isrFuncPointer, uint32
serviceReqPrioNumber);
 * API for Interrupt handler install for SW Managed interrupts.
 * This API installs the isr to SW interrupt vector.
 * This must be used only when IFX_USE_SW_MANAGED_INT is defined in
Ifx_Cfg.h
 *
 * \param isrFuncPointer pointer to ISR function.
 * \param serviceReqPrioNumber ISR priority.
 */

IfxCpu_Irq_installInterruptHandler(Stm0_Isr, STM0_ISR_PRIORITY);
```

6. Configure the STM peripheral for compare interrupts as shown in the function below:

```
// File: <application file>
void Initialize_StmTicks(void)
{
    IfxStm_CompareConfig stmCompareConfig;
    // suspend by debugger enabled
    IfxStm_enableOcdsSuspend(&MODULE_STM0);

    //Call the constructor of configuration
    IfxStm_initCompareConfig(&stmCompareConfig);
    //Modify only the number of ticks and enable the trigger output
    stmCompareConfig.ticks           = 1000; /*Interrupt after 1000 ticks
from now */
    stmCompareConfig.triggerPriority = STM0_ISR_PRIORITY;
    stmCompareConfig.typeOfService   = IfxSrc_Tos_cpu0;

    //Now Compare functionality is initialized
    IfxStm_initCompare(&MODULE_STM0, &stmCompareConfig);
}
```

Note: *Configuration service request node is done by the function IfxStm_initCompare as used in above code snippet. With this function step 2 and 3 are not needed.*

3.9 Debugging with Traps

Trap mechanism in TriCore™ works in similar way as interrupts. However, ***Base Template Project***s implemented this little differently to make trap vector table statically located.

Base Template Projects implement following features to make it simple for user and yet flexible enough to debug and configure required functionality.

- All trap vectors for every TriCore™ CPUs are made available
- Trap information could be traced with variable "**trapWatch**"

- Application could implement a call-back hook function for every trap

Following sections provide information about the features in detail.

3.9.1 Tracing with Trap Information

When the trap occurs, CPU stores the trace information in different registers. With ***iLLD*** you can trace trap information with variable "**trapWatch**". This variable is an internal variable inside the function, but passed to call-back hook function as parameter. This variable is an instance of structure type :

```
//Type define form file: IfxCpu.h:

typedef struct
{
    unsigned int tAddr;          /**< Trap address*/

    unsigned int tId : 8;        /**< Trap ID*/
    unsigned int tClass : 8;     /**< Trap class*/
    unsigned int tCpu : 3;       /**< Host CPU*/

} IfxCpu_Trap;
```

Following table provide the information related to the data members of the above structure:

Table 19 Trap information structure type

Element	Description
tAddr	Address at which trap occurred ***Attention:*** ***The address of asynchronous trap may not point to the instruction which caused the trap***
tId	Trap identification number
tClass	Trap class
tCpu	Host CPU which is affected by trap

Tip: You could watch the above variable in the debugger trace window as a local variable.

3.9.2 Trap Call-back Hook Functions

Trap callback hook functions are the functions/ macros which are invoked by the TSR. You could implement the Trap callback hook functions as an extension according to the need of application for an individual trap. These hooks are unique for each trap but common for all CPUs except for SysCall. For SysCall trap ***iLLD*** implements unique callback hook for each CPU.

Following table provide the set of hooks available to you through ***iLLD***, which is part of ***Base Template Project***s.

Table 20

Call back hook	is function call allowed	Description
IFX_CFG_CPU_TRAP_MME_HOOK(trapWatch)	Yes	Memory Management Error
IFX_CFG_CPU_TRAP_IPE_HOOK(trapWatch)	Yes	Internal Protection Error
IFX_CFG_CPU_TRAP_IE_HOOK(trapWatch)	Yes	Instruction Error
IFX_CFG_CPU_TRAP_CME_HOOK(trapWatch)	No	Context Management Error
IFX_CFG_CPU_TRAP_BE_HOOK(trapWatch)	Yes	System Bus and Peripheral Error
IFX_CFG_CPU_TRAP_ASSERT_HOOK(trapWatch)	Yes	Assertion Traps
IFX_CFG_CPU_TRAP_SYSCALL_CPU**x**_HOOK(trapWatch)	Yes	System Call Traps for CPU**x**. Where x is the index of CPU. (e.g. for TC39x 6 cpus with index 0 till 5)
IFX_CFG_CPU_TRAP_NMI_HOOK(trapWatch)	Yes	Non Maskable Interrupt

The parameter, that is passed in all of the above traps is of type "**IfxCpu_Trap**" as explained in previous section. by default these callback hooks are defined to empty. You could override this default definition with your own application call as in the example code below.

```
// Application file:

void assertTrap(IfxCpu_Trap trapWatch)
{
    uint8 hostCpu= trapWatch.tCpu;
    uint8 trapClass= trapWatch.tClass;
      // and so on..
}
```

You must also define the hook as below:

```
 //file: 0_Src/AppSw/CpuGeneric/Config/Ifx_Cfg.h:

extern void assertTrap(IfxCpu_Trap trapWatch);

#define IFX_CFG_CPU_TRAP_ASSERT_HOOK(trapWatch) assertTrap(trapWatch)
```

3.10 Using End-Init APIs

End-Init functionality is provided by peripherals to avoid accidental update of critical registers. Which means, these registers are initialized once and they are protected at the end-of-initialization. End-Init protection feature is implemented as sub function of watchdog peripheral.

With AURIX™ microcontrollers, you have a watchdog instance for each TriCore™ CPU and safety watchdog. With AURIX™ 2ndGeneration controllers, additionally a global watchdog is implemented. A register could be protected with one or more of such watchdog's End-Init protection functionality as below

- **CEy**: CPU critical registers. Writable only when CPUy WDT ENDINIT=0 (y=CPU number)
- **E**: System critical registers - Writable when any (one or more) CPUy WDT ENDINIT=0 or EICON0.ENDINIT =0
- **SE**: Safety critical registers - Writable only when Safety WDT ENDINIT=0 or SEICON0.ENDINIT=0

All the registers don't have such protection. You shall refer to hardware documentation if they are protected and what kind of End-Init protection. Modification of such registers need following sequence to be followed:

1. Disable End-Init protection
2. Modify the register/s within the watchdog timeout period
3. Enable End-Init protection

iLLDs provide End-Init protection enable/ disable functions with file : IfxScuWdt.h. Following sections introduce you to the ***API***s as briefly:

3.10.1 APIs to get the current passwords

These ***API***s fetch the current password. When the password is read from hardware register, the value is inverted. This ***API*** inverts the value after read and this value could be directly used while accessing the watchdog/End-Init registers. Following are the ***API***s for different watchdogs/ End-Init hardware interfaces:

```
/** \brief Prototype for API to fetch current password of CPU Watchdog module.
 */
IFX_EXTERN uint16 IfxScuWdt_getCpuWatchdogPassword(void);
```

```
/** \brief Prototype for API to fetch current password of global endinit
Watchdog module.
 */
IFX_EXTERN uint16 IfxScuWdt_getGlobalEndinitPassword(void);
```

```
/** \brief Prototype for API to fetch current password of Safety Watchdog
module.
 */
IFX_EXTERN uint16 IfxScuWdt_getSafetyWatchdogPassword(void);
```

```
/** \brief Prototype for API to fetch current password of global safety
endinit Watchdog module.
 */
IFX_EXTERN uint16 IfxScuWdt_getGlobalSafetyEndinitPassword(void);
```

Please refer to ***iLLD*** documentation (or doxygen comments at the source files) to fetch more information about the ***API***s above.

3.10.2 APIs to clear End-Init protection

These ***API***s clear End-Init protection of the registers. Following are the ***API***s for different watchdogs/ End-Init hardware interfaces:

```
/** \brief Prototype for API to Clear ENDINIT bit provided by CPU WDT Hardware
module.
 */
IFX_EXTERN void IfxScuWdt_clearCpuEndinit(uint16 password);
```

```
/** \brief Prototype for API to Clear global ENDINIT bit provided by CPU WDT
Hardware module.
 */
IFX_EXTERN void IfxScuWdt_clearGlobalEndinit(uint16 password);
```

```
/** \brief Prototype for API to Clear ENDINIT bit provided by Safety WDT
Hardware module.
 */
IFX_EXTERN void IfxScuWdt_clearSafetyEndinit(uint16 password);
```

```
/** \brief Prototype for API to Clear global safety ENDINIT bit provided by
safety WDT Hardware module.
 */
IFX_EXTERN void IfxScuWdt_clearGlobalSafetyEndinit(uint16 password);
```

Please refer to ***iLLD*** documentation (or doxygen comments at the source files) to fetch more information about the ***API***s above.

3.10.3 APIs to set End-Init protection

These ***API***s set End-Init protection of the registers. Following are the ***API***s for different watchdogs/ End-Init hardware interfaces:

```
/** \brief Prototype for API to set ENDINIT bit provided by CPU WDT Hardware
module.
 */
IFX_EXTERN void IfxScuWdt_setCpuEndinit(uint16 password);
```

```
/** \brief Prototype for API to set global ENDINIT bit provided by CPU WDT
Hardware module.
 */
IFX_EXTERN void IfxScuWdt_setGlobalEndinit(uint16 password);
```

```
/** \brief Prototype for API to Set ENDINIT bit provided by Safety WDT
Hardware module.
 */
IFX_EXTERN void IfxScuWdt_setSafetyEndinit(uint16 password);
```

```
/** \brief Prototype for API to set global safety ENDINIT bit provided by
safety WDT Hardware module.
 */
IFX_EXTERN void IfxScuWdt_setGlobalSafetyEndinit(uint16 password);
```

3.10.4 Example usage of End-Init APIs

Below example provide the sequence of ***API***s to be called in your application while accessing a register which is protected by safety End-Init:

```
/* Include the IfxScuWdt.h to access the End-init APIs */
#include "IfxScuWdt.h"
#include "Ifx_Types.h"



void ExampleFunction_endInit(void)
{

    uint16 safetyWdtPassword= IfxScuWdt_getSafetyWatchdogPassword();

    /* clear endinit protection */
    IfxScuWdt_clearSafetyEndinit(safetyWdtPassword);

    /* Your code here: as an example to access SYS PLL register */
    SCU_SYSPLLCON0.B.MODEN= (uint32)IfxScuCcu_ModEn_enabled;

    /* set the endinit protection again */
    IfxScuWdt_setSafetyEndinit(safetyWdtPassword);
}
```

3.11 Performance Measurement

TriCore™ provides feature to measure the CPU performance with set of counters. These counters are accessible during debug mode, i.e. when CPU_DBGSR.DE= 1 (which is also set when the debuggers are connected).
Below table provides the list of counters provided by an instance of TriCore™ CPU:

Table 21 Performance Control Registers

Register	**Description**
CCTRL	Counter control register
CCNT	CPU clock count register
ICNT	CPU instruction count register
M1CNT	Multi count register 1
M2CNT	Multi count register 2
M3CNT	Multi count register 3

3.11.1 Data Structure

Clock performance counter registers are represented by a data structure type **IfxCpu_Perf** as shown below:

```
typedef struct
{
    IfxCpu_Counter instruction;        /**< \brief Instruction counter */
    IfxCpu_Counter clock;              /**< \brief CPU clock counter */
    IfxCpu_Counter counter1;           /**< \brief Multi counter 1 */
    IfxCpu_Counter counter2;           /**< \brief Multi counter 2 */
    IfxCpu_Counter counter3;           /**< \brief Multi counter 3 */
} IfxCpu_Perf;
```

Where each counter is represented by data structure type **IfxCpu_Counter** as shown below:

```
typedef struct
{
    uint32  counter;          /**< \brief Counter value */
    boolean overlfow;         /**< \brief sticky overlfow */
} IfxCpu_Counter;
```

CPU performance counter ***API***s would be able to extract the information with data structure formats as above

3.11.2 Performance Measurement APIs

Performance counters are configured and accessed by following ***API***s, which are available through IfxCpu.h.
Reset and Start counters ***API*** would configure the performance counters and trigger them to start. The prototype of this ***API*** is as shown in code segment below

```
/** \brief Reset and start instruction, clock and multi counters
 *
 * Reset and start CCNT, ICNT, M1CNT, M2CNT, M3CNT. the overflow bits are
cleared.
 * \param mode Counter mode
 * \return None
 */
IFX_INLINE void IfxCpu_resetAndStartCounters(IfxCpu_CounterMode mode);
```

Stop Counters ***API*** would stop the counter first and return the values of all counters as data structure format **IfxCpu_Perf**. Prototype of the ***API*** is shown as in the code block below:

```
/** \brief Stop counters, return their values
 *
 * Stop CCNT, ICNT, M1CNT, M2CNT, M3CNT and return their values;
 *  \Note The CCTRL is reset to 0, for more accurate measurements and has to
be initialized again before strating the next performance measurement.
 * \return Performance counter result
 */
IFX_INLINE IfxCpu_Perf IfxCpu_stopCounters(void);
```

3.11.3 Example

Below is an example of usage of performance measurement ***API***s. In this example, performance is measured for execution of function **main_Dhrystone**.

```
#include "IfxCpu.h"

/* Instantiate global data structures */
IfxCpu_Perf perf;

/* Variable for cpu clock count */
float miCnt;

void ExampleFunction_perfMeasure(void)
{
    /* Timers Start */
    IfxCpu_resetAndStartCounters(IfxCpu_CounterMode_normal);

    /* Your code here for performance measurement */
     main_Dhrystone();

    /* Timers Stop */
    perf= IfxCpu_stopCounters();
    miCnt= perf.clock.counter / 1000000.0;
}
```

4 Merging Examples to Base Template

When you install ***Base Template Project***s, you get a folder with folder name: "**_Example_<TC3xx>**". This folder contains several sub-folders, each of them are sub set of project source files and/or project configuration files. By the folder-name of such sub-folders, you could figure out whether an example is directly compatible with targeted template project. As examples, if the folder-name ends with <TC3xx>, the example could be used with any of the ***Base Template Project***s (each corresponds to a derivate). If a folder-name ends with TC39x then such example could be used with ***Base Template Project***s of <TC3xx>.

This section details how to merge the examples with the ***Base Template Project***s with Eclipse CDT environment and outside Eclipse environment (normally for non Eclipse users).

4.1 Working Outside Eclipse Environment

Following are the steps to build project, when you want to do the merge examples outside Eclipse environment:

1. At your workspace, copy the folder tree of BaseFramework_<TC3xx> project (you must be sure that this project is correctly setup for used compiler, is buildable successfully and it runs on the hardware)
2. Paste it as new folder-name which is meaningful for the application you are going to work on (e.g. MyApplication_TC3xx)

   Tip: *Always suffix the root folder name of project to represent the microcontroller derivate name and its step,because this information is not easily found from the source files/ configuration inside your project.*

3. Merge and overwrite the folder tree at project root level, with **_Examples_<TC3xx>/<name of example>_<suffix for derivate>** folder (overwrite the files/folders which are same). Take care to copy the folder structure one to one.

Now your example is ready to be used

4.2 Working within Eclipse Environment

Following are the steps to build project when you want to merge with Eclipse environment:

Copy the working template project

1. Under eclipse Project explorer: Copy BaseFramework_TC3xx project with **Ctrl+c** keys (or right-mouse-click on project root folder and click **Copy**)

   Attention: ***You need to be sure that this project is correctly setup for used compiler, is buildable successfully and it runs on the hardware.***

2. Paste **Ctrl+v** (or right-mouse-click at the end of folders can click **Paste**). You need to provide a new name in the window, which comes up. Give any new name which meaningful for the application you target (e.g. MyApplication_TC3xx). This is new project which has all the required settings which you have already done in template project.

Merge the example sources:

1. Right-click on newly copied project root folder (e.g. MyApplication_TC3xx) at eclipse project explorer, and click on **"Import"** as shown in the figure below:

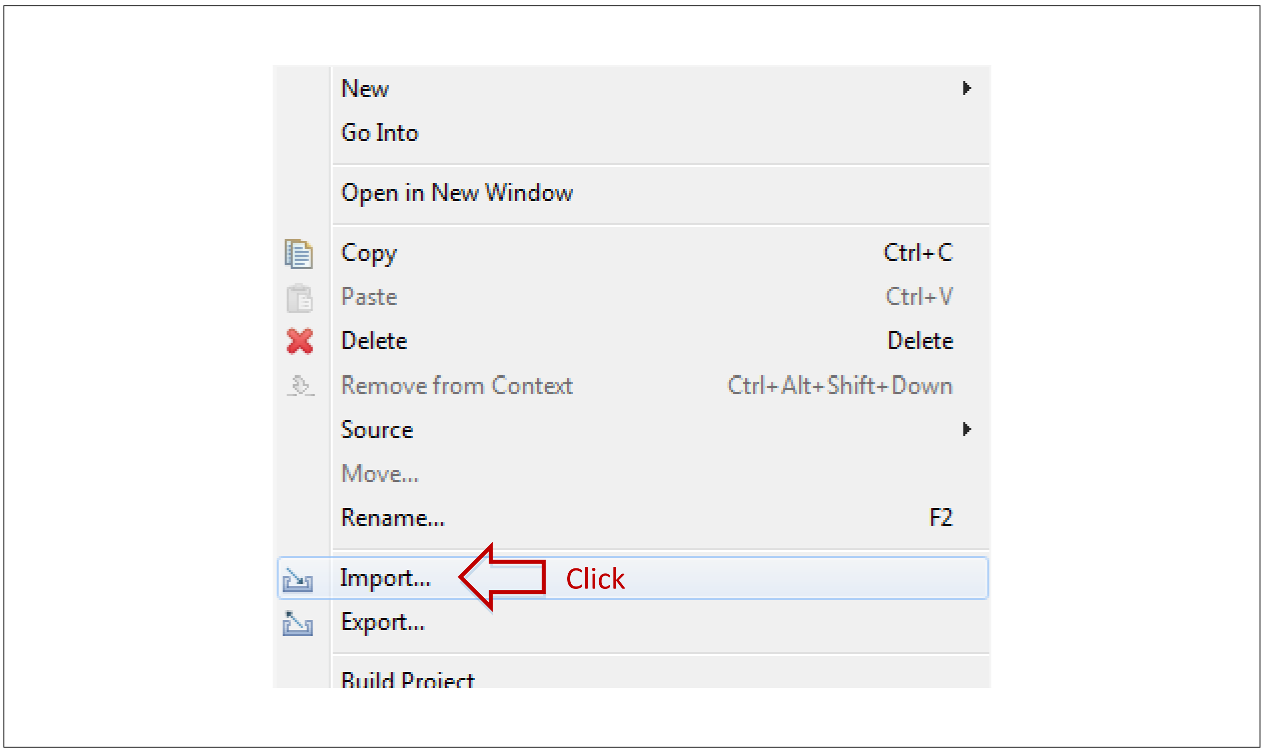

Figure 4 Import Menu

2. A window **“Import”** opens up, under **“Select an import source:”** expand the tab **“General”** and click on **“File System”** and click **“Next”** button as shown below:

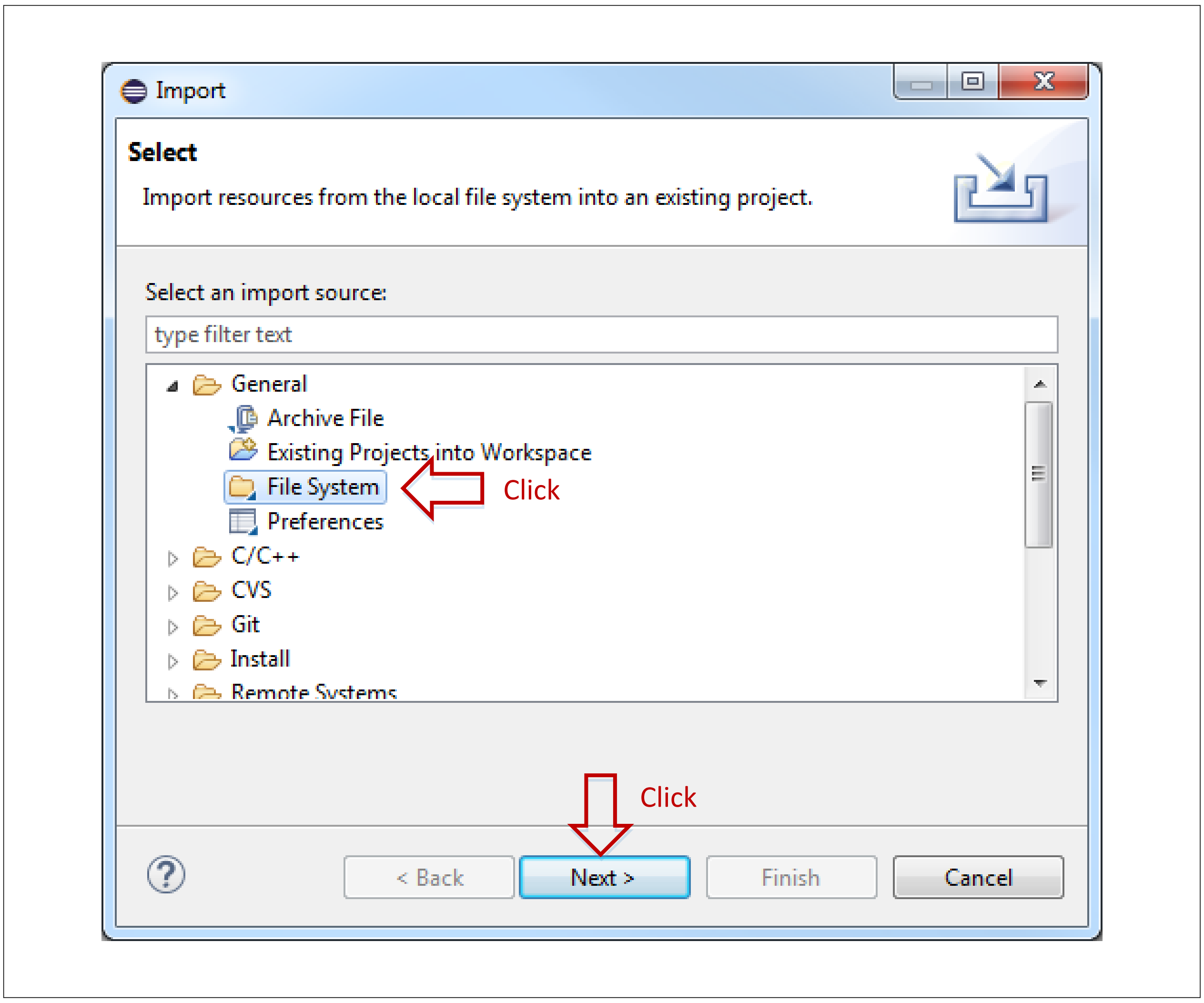

Figure 5 Import Wizard

3. Now this change to window as shown below:

Note: *Below is the generic picture to show example folder as <Example Name> and project name, that is merge target, as <Your Project>. This looks little different with your eclipse window for your project and the example you are selecting (<Example Name> → <name of example>_<suffix for derivate>).*

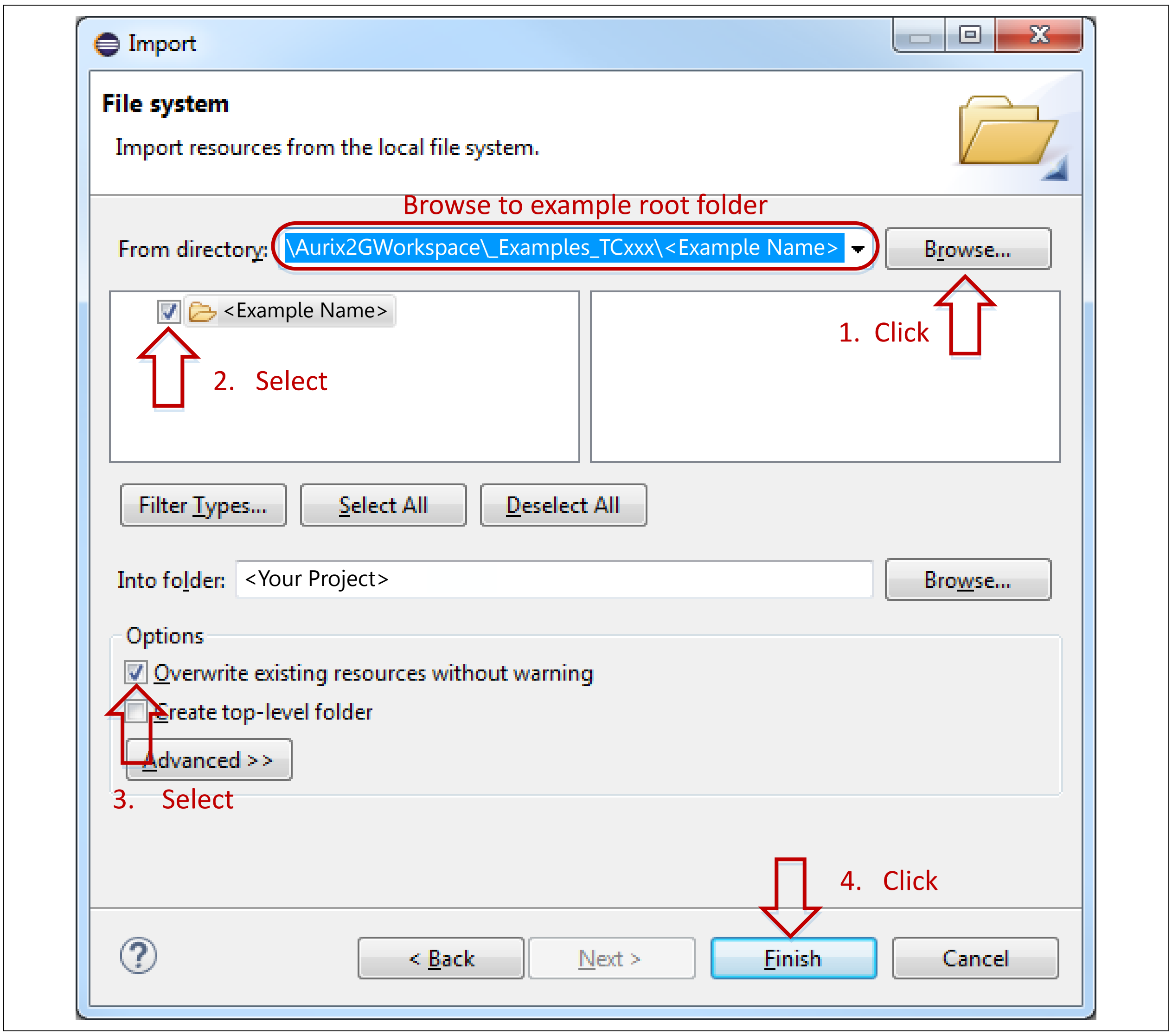

Figure 6 Import Folders

4. Click on **"Browse"** button to browse to the workspace, where the template projects are installed and select the folder **_Examples_<TC3xx>/<name of example>_<suffix for derivate>**.
5. Path for folder: <name of example>_<suffix for derivate> appears in **"From directory:"** field as above figure.
6. Select the check-box against <name of example>_<suffix for derivate> in the selection field.
7. Also check the check-box under **"Options"**, "**Overwrite existing resources without warning**"
8. Click "**Finish**"
9. Now example folders are merged and the example is ready to be used

5 Migrating from Software Framework To BIFACES

You have already existing projects which were using Software Framework Tools as build environment. Now you would like to migrate to ***BIFACES***. This section provides the information, what is necessary for such migration.

You might have one or more of the following scenarios.

- Import the entire project to ***BIFACES*** tools and build it (This is not migration to ***Base Template Project***s, but use the ***BIFACES*** tools as build environment)
- Migrate to ***BIFACES*** while keeping old folder Structure of iLLD (where: iLLD version < V1.0.0.12.0)

- Migrate only application Files to ***Base Template Project***s

5.1 Import the Entire Project to BIFACES

In this scenario, you would like to work with backward compatibility mode of ***BIFACES***. Please note that, this works only for basic use cases of Software Framework build tools. Complex usage is not tested. This is not guaranteed to support this for all upcoming releases of ***BIFACES***.

Following are the steps:

- Launch your project in the ***BIFACES*** workspace.
- Start build
- Build outputs get generated
- Output folders are arranged according to ***Base Template Project*** folder structure

Note: You will notice following message from the build environment:

```
make all
Now the make files are generated !!!
 ______________________________________________________________________
!!!                            IMPORTANT                            !!!
! You are using BIFACES with a project that was made for               !
! Software Framework!                                                  !
! Please note that, all features may not work.                         !
! Backward compatibility for Software Framework would be removed with  !
! next releases of BIFACES.                                            !
!______________________________________________________________________!

Now Build started !!!
```

5.2 Migrate to BIFACES while Keeping Old Folder Structure of iLLD

With this scenario you (probably) don't want to make any changes in the source code and the folder structure. The folder structure remains as defined by iLLD version which was earlier to V1.0.0.12.0. Software Framework Projects are configured to build sources with such folder structure where 'default' include paths are adjusted as required. To migrate you need to do a "small" update to a configuration change once migration is done. Following are the steps.

- Copy the working ***Base Template Project*** as described at section: ***Copy Template Project as My Application***
- Delete "**0_Src**" folder
- Copy the "**0_Src**" folder form the source project
- Open file "**Config.xml**", from folder: "**1_ToolEnv/0_Build/1_Config**"
- Look for the xml element "**specificInclude**" under root and modify the attribute "**internalPaths**" to upend with file pattern: "*****" as shown below

```
<!--  File: Config.xml -->
..
..
<specificInclude internalPaths= '*/Sfr/*, */BaseSw/*/Tricore, */BaseSw/*/
Platform, */BaseSw/*/CpuGeneric, *' />
```

- Build the project
- Build outputs get generated
- Output folders are arranged according to ***Base Template Project*** folder structure

Note: The added file pattern "" results in additional include paths, which adds all folders under source folder tree as include path. This has negative impact on performance related to build time*

5.3 Migrate only Application Files to BIFACES Template Projects

This is the simple case where you have the possibility to update the iLLD and project folder structure to latest iLLD requirements. As a pre-requisite to the migration steps below you need to update to iLLD version which is V1.0.0.12.0 or later.

- Merge the AppSw contents
- Build the project
- Build outputs get generated
- Output folders are arranged according to ***Base Template Project*** folder structure

Glossary

API
Application **P**rogram **I**nterface

Architecture
CPU Architecture. e.g. TriCore™, Arm, Intel e.t.c

Base Template Project
This is a ***BIFACES Project***, which is provided for each micro-controller derivative. This project contains

- Basic project configurations,
- Startup, SFR Headers and infrastructure driver files
- Template linker command files for each supported compiler.

The infrastructure drivers used with base projects are respective ***iLLD***s of corresponding microcontroller detivate. User could enhance this project with own application code.

BIFACES
Build and **I**ntegration **F**ramework for **A**utomotive **C**ontroller **E**mbedded **S**oftware

BIFACES Project
It is a project environment or working directory, where source files and/or configuration files are stored. Files and folders in such working directory is organized as required by ***BIFACES***. ***BIFACES*** works only on such a working directory.

Filename Pattern
Strings with wild-card characters (such as '*****' and '**?**') to select set of files which match the pattern. Multiple characters are selected with '*****' and single character is selected with '**?**'.

iLLD
Infineon **L**ow **L**evel **D**river

Option Set
Set of command-line options to control the behavior of ***Tool***s during build process. E.g. Compiler, Assembler, Linker, Archiver e.t.c.

Output

Result of build process for a ***Target***. e.g. Elf file for TriCore™ ***Target***, Library archive file for Intel CPU e.t.c.

Primary Architecture
Default ***Architecture***of the project. It is possible to configure a project with multiple ***Architecture***s. Files are explicitly associated with one of the available ***Architecture***s. In cases, where a file is not associated to any of the ***Architecture***s, ***BIFACES*** automatically associates such a file to this default ***Architecture***.

Primary Option Set
Default ***Option Set*** of the ***Tool***. It is possible to configure a ***Tool*** with multiple ***Option Set***s. Files are explicitly associated with one of the those ***Option Set***s. In cases, where a file is not associated to any of the ***Option Set***s, ***BIFACES*** automatically associates such a file to this default ***Option Set***.

Primary Target
Default ***Target*** of the ***Architecture***. It is possible to configure an ***Architecture*** with multiple ***Target***s. Files are explicitly associated with one of the those ***Target***s. In cases, where a file is not associated to any of the ***Target***s, ***BIFACES*** automatically associates such a file to this default ***Target*** .

Primary Toolchain
Default ***Toolchain*** of the ***Architecture***. It is possible to configure an ***Architecture*** with multiple ***Toolchain***s. Files are explicitly associated to one of the ***Tool***s under one of such ***Toolchain***s. In cases, where a file is not associated to any of the ***Tool***s, ***BIFACES*** automatically associates such a file to an appropriate ***Tool*** (based on file type), which is under such ***Toolchain***.

Target
A Build Target. This could be an executable which is targeted to run on single or group of CPU instance/s Or this could be a library which is an archive of object files built for an ***Architecture*** Or simply an image of variables/ constants.

Tool
Tool to build source files. Normally a tool is used for a particular phase of build process e.g. Compiler, Assembler, Linker, Archiver e.t.c. Tools could be internal or external.

Toolchain
Collection of ***Tool***s, for a specific ***Architecture***, to build source files. e.g. GNU C from Hightec, Tasking from Altium, Diab from Windriver e.t.c.

Revision history

Document version	Date of release	Description of changes
1.0.1	22.11.2017	• Updates to the chapter ***Merging Examples to Base Template*** • Section "***Migrating from Software Framework To BIFACES*** added
1.0.0	20.07.2017	• Initial creation

Edition 2018-07-12

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference IFX-xew1496844093459