Processor Expert User Guide

Revised: 13 March 2014

How to Contact Us

Corporate Headquarters	Freescale Semiconductor, Inc. 6501 William Cannon Drive West Austin, TX 78735 U.S.A.
World Wide Web	`http://www.freescale.com/codewarrior`
Technical Support	`http://www.freescale.com/support`

Table of Contents

1 Introduction 7

Overview 7
Features of Processor Expert 8
  Key Components 10
  Advantages 10
Concepts 13
  Embedded Components 13
  Creating Applications 14
  RTOS Support 15
Terms and Definitions Used in Processor Expert 15

2 User Interface 19

Main Menu 20
  Project Pop-up Menu 20
  Processor Expert Options 20
Components View 23
  View Menu 26
  Pop-up Menu 26
Components Library View 28
  Modes 28
  Filtering 28
  Pop-up Menu 28
  Component Assistant 29
Component Inspector View 30
  Read Only Items 33
  View Mode Buttons 33
  View Menu 33
  Pop-up Menu 34
  Inspector Items 35
  Items Visibility 38
  Pin Settings 39
  Component Inspector 39

Configuration Inspector 48
Processor View 49
Control Buttons 50
Memory Map View 53
Configuration Registers View 54
Initialization Sequence View 55

3 Application Design 51

Creating Application using Processor Expert 51
Basic Principles 52
Embedded Components 52
Processor Components 60
Configuring Components 73
Interrupts and Events 73
Configurations 77
Design Time Checking: Consequences and Benefits 78
Timing Settings 79
Creating User Component Templates 81
Signal Names 82
Component Inheritance and Component Sharing 84
Pin Sharing 85
Implementation Details 95
Reset Scenario with PE for ColdFire and Kinetis Microcontrollers 96
Version Specific Information for Kinetis and ColdFire+ 97
Code Generation and Usage 98
Code Generation 98
Predefined Types, Macros and Constants 102
Typical Usage of Component in User Code 108
User Changes in Generated Code 114
Embedded Component Optimizations 115
General Optimizations 115
General Port I/O Optimizations 116
Timer Components Optimizations 117
Code Size Optimization of Communication Components 118
Converting Project to Use Processor Expert 119

Low-level Access to Peripherals 120
Peripheral Initialization 120
Peripheral Driver Implementation 120
Physical Device Drivers 120
Processor Expert System Library 121
Direct Access to Peripheral Registers 122
Processor Expert Files and Directories 124
PE Project File 124
Project Directory Structure 125
User Templates and Components 125
Static Code Support in Processor Expert 125
Static Initialization Code 125
Standalone and Linked Project 126
Standalone Mode 127
Linked Mode 127
Processor Expert Static Code Repository 127
Project Static Code Directory Structure 128
Project Static Modules 128
Processor Component Files 129
Peripheral Initialization Component Files 129
Peripheral Memory Map Files 130
Common Project Files 130
Processor Expert Static Code Repository and Project Repository Synchronization 131
Static Code Track Changes 132
Multiple Properties Accessing Common Register and <Automatic> Value 133
Internal signals 133
Differentiation of Internal Signals and Pins 136
Internal signal producer component linking 138

4 Processor Expert Tutorials 141

Tutorial Project 1 for Kinetis Microcontrollers 141
Creating a New Project 141
Adding Components 142
Configuring Components 143

Code Generation 144

Index **149**

1 Introduction

Processor Expert (PE) is designed for rapid application development of embedded applications for a wide range of microcontrollers and microprocessor systems.

This chapter explains:

- Overview
- Features of Processor Expert
- Concepts
- Terms and Definitions Used in Processor Expert

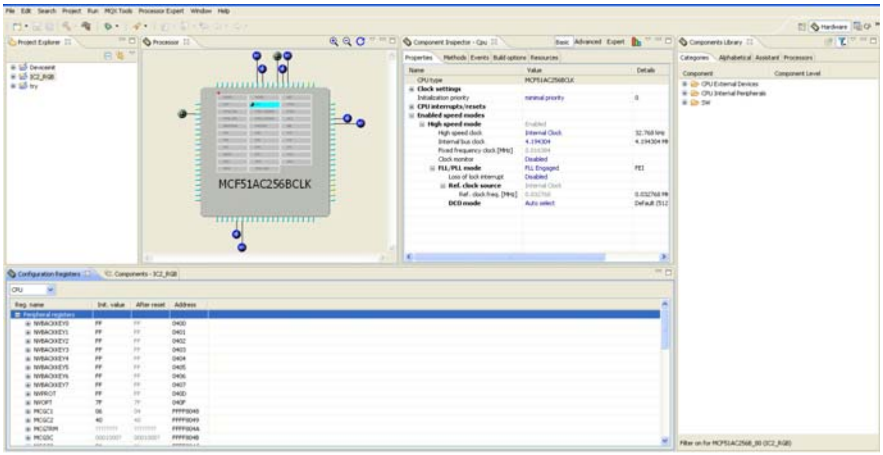
Overview

Processor Expert provides an efficient development environment for rapid application development of the embedded applications. You can develop embedded applications for a wide range of microcontrollers and microprocessor systems using Processor Expert.

Processor Expert is integrated as a plug-in into the Eclipse IDE. You can access Processor Expert from the IDE using the Processor Expert menu in the IDE menu bar. The Processor Expert plug-in generates code from the embedded components and the IDE manages the project files, and compilation and debug processes.

The figure below shows the **Processor Expert** menu in the IDE menu bar.

Figure 1.1 IDE with Processor Expert Menu


File Edit Search Project Run MQX Tools Processor Expert Window Help
Project Explorer
Processor
MCF51AC256BCLK
Component Inspector - Cpu
Basic Advanced Expert
Properties Methods Events Build options Resources
Components Library
Categories Alphabetical Assistant Processors
Component Component Level
CPU External Devices
CPU Internal Peripherals
SW
Configuration Registers
Components - IC2_RGB
CPU

Name	Value	Details
CPU type	MCF51AC256BCLK	
Clock settings		
Initialization priority	minimal priority	0
CPU interrupts/resets		
Enabled speed modes		
High speed mode	Enabled	
High speed clock	Internal Clock	32.768 kHz
Internal bus clock	4.194304	4.194304 MH
Fixed frequency clock [MHz]	0.016384	
Clock monitor	Disabled	
FLL/PLL mode	FLL Engaged	FEI
Loss of lock interrupt	Disabled	
Ref. clock source	Internal Clock	
Ref. clock freq. [MHz]	0.032768	0.032768 MH
DCO mode	Auto select	Default (512

Reg. name	Init. value	After reset	Address
Peripheral registers			
NVBACKKEY0	FF	FF	0400
NVBACKKEY1	FF	FF	0401
NVBACKKEY2	FF	FF	0402
NVBACKKEY3	FF	FF	0403
NVBACKKEY4	FF	FF	0404
NVBACKKEY5	FF	FF	0405
NVBACKKEY6	FF	FF	0406
NVBACKKEY7	FF	FF	0407
NVPROT	FF	FF	040D
NVOPT	7F	7F	040F
MCGC1	06	04	FFFF8048
MCGC2	40	40	FFFF8049
MCGTRM	????????	????????	FFFF804A
MCGSC	00010001	00010001	FFFF804B

NOTE	For more information about how to create a new project, refer to the Processor Expert Tutorials chapter or Creating Application using Processor Expert chapter for step-by-step instructions on how to create a new Processor Expert project.

NOTE	Processor Expert generates all drivers during the code generation process. The generated files are automatically inserted into the active (default) target in the project. For more information on generated files, refer to the Code Generation chapter.

Features of Processor Expert

Processor Expert has built-in knowledge (internal definitions) about all microcontroller units and integrated peripherals. The microcontroller units and peripherals are encapsulated into configurable components called embedded components, each of which provides a set of useful properties, methods, and events.

The following are the main features of Processor Expert:

- The application is created from components called embedded components.
- Embedded components encapsulate functionality of basic elements of embedded systems like processor core, processor on-chip peripherals, FPGA, standalone peripherals, virtual devices, and pure software algorithms, and change these facilities to properties, methods, and events (like objects in OOP).

- Processor Expert connects, and generates the drivers for embedded system hardware, peripherals, or used algorithms. This allows you to concentrate on the creative part of the whole design process.
- Processor Expert allows true top-down style of application design. You can start the design directly by defining the application behavior.
- Processor Expert works with an extensible components library of supported microprocessors, peripherals, and virtual devices.
- Processor Expert peripheral initialization components generate effective initialization code for all on-chip devices and support all their features.
- Logical Device Drivers (LDD components) are efficient set of embedded components that are used together with RTOS. They provide a unified hardware access across Microcontrollers allowing to develop simpler and more portable RTOS drivers or bare board application. For more details, refer to the Logical Device Drivers topic.
- Processor Expert allows to examine the relationship between the embedded component setup and control registers initialization.

An intuitive and powerful user interface allows you to define the system behavior in several steps. A simple system can be created by selecting the necessary components, setting their properties to the required values and also dragging and dropping some of their methods to the user part of the project source code.

The other key features are:

- Design-time verifications
- Microcontroller selection from multiple Microcontroller derivatives available
- Microcontroller pin detailed description and structure viewing
- Configuration of functions and settings for the selected Microcontroller and its peripherals
- Definition of system behavior during initialization and at runtime
- Design of application from pre-built functional components
- Design of application using component methods (user callable functions) and events (templates for user written code to process events, e.g. interrupts)
- Customization of components and definition of new components
- Tested drivers
- Library of components for typical functions (including virtual SW components)
- Verified reusable components allowing inheritance
- Verification of resource and timing contentions
- Concept of project panel with ability to switch/port between Microcontroller family derivatives

- Code generation for components included in the project
- Implementation of user written code
- Interface with Freescale CodeWarrior

This section includes the following topics:

- Key Components
- Advantages

Key Components

The key components are:

- Graphical IDE
- Built-in detailed design specifications of the Freescale devices
- Code generator

Advantages

PE based tool solution offers the following advantages to Freescale Microcontroller customers:

- In all phases of development, customers will experience substantial reductions in
  - development cost
  - development time
- Additional benefits in product development process are:
  - Integrated Development Environment Increases Productivity
  - Minimize Time to Learn Microcontroller
  - Rapid Development of Entire Applications
  - Modular and Reusable Functions
  - Easy to Modify and Port Implementations

Integrated Development Environment Increases Productivity

Integrated development environment increases productivity:

- This tool lets you produce system prototypes faster because the basic setup of the controller is easier. This could mean that you can implement more ideas into a prototype application having a positive effect on the specification, analysis, and design phase. Processor Expert justifies its existence even when used for this purpose alone.

- This system frees you up from the hardware considerations and allows you to concentrate on software issues and resolve them.
- It is good for processors with embedded peripherals. It significantly reduces project development time.

The primary reasons why you should use Processor Expert are:

- Processor Expert has built-in knowledge (internal definition) of the entire microcontroller with all its integrated peripherals.
- Processor Expert encapsulates functional capabilities of microcontroller elements into concepts of configurable components.
- Processor Expert provides an intuitive graphical user interface, displays the microcontroller structure, and allows you to take the advantage of predefined and already verified components supporting all typically used functions of the microcontroller.
- Applications are designed by defining the desired behavior using the component settings, drag and drop selections, utilizing the generated methods and events subroutines, and combining the generated code with user code.
- Processor Expert verifies the design based on actual microcontroller resource and timing contentions.
- Processor Expert allows the efficient use of the microcontroller and its peripherals and building of portable solutions on a highly productive development platform.

Minimize Time to Learn Microcontroller

There are exciting possibilities in starting a new project if the user is starting from ground zero even if the user is using a new and unfamiliar processor.

- You can work on microcontroller immediately without studying about the microcontroller
- Documentation
- You can implement simple applications even without deep knowledge of programming
- PE presents all necessary information to the user using built-in descriptions and hints
- PE has built-in tutorials and example projects.

Rapid Development of Entire Applications

Processor Expert allows you to try different approaches in real time and select the best approach for the final solution. You are not confined to a pre-determined linear approach to a solution.

- Easy build of application based on system functional decomposition (top-down approach)
- Easy microcontroller selection
- Easy Processor initialization
- Easy initialization of each internal peripheral
- Simple development of reusable drivers
- Simple implementation of interrupt handlers
- Inherited modularity and reuse
- Inherited ease of implementation of system hardware and software/firmware modifications

Modular and Reusable Functions

Processor Expert decreases the start-up time and minimizes the problems of device.

- It uses the concept of a function encapsulating entity called embedded component with supporting methods and events
- Uses a library of predefined components
- Uses the concept of device drivers and interrupt handlers that are easy to reapply
- Uses the concept of well-documented programming modules to keep the code well organized and easy to understand

NOTE Processor Expert embedded component were formerly called Processor Expert Embedded Beans.

Easy to Modify and Port Implementations

Processor Expert allows optimal porting to an unused processor.

- Supports multiple devices within a project and makes it extremely easy to switch them
- Supports desired changes in the behavior of the application with an instant rebuild
- Supports interfacing of the IDE

Concepts

The main task of Processor Expert is to manage processor and other hardware resources and to allow virtual prototyping and design.

Code generation from components, the ability to maintain user and generated code, and an event based structure significantly reduce the programming effort in comparison with classic tools.

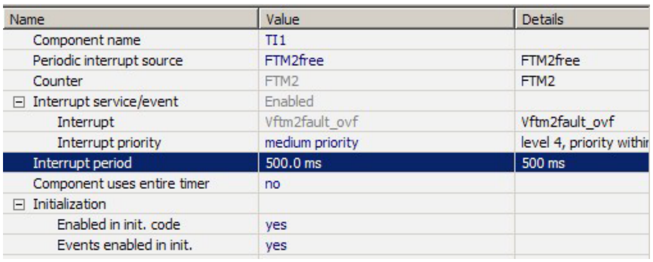
This section covers the following topics:

- Embedded Components
- Creating Applications
- RTOS Support

Embedded Components

Component is the essential encapsulation of functionality. For instance, the *TimerInt* component encapsulates all processor resources that provide timing and hardware interrupts on the processor.

Figure 1.2 Example of TimerInt Component (Periodical Event Timer) Properties



Name	Value	Details
Component name	TI1	
Periodic interrupt source	FTM2free	FTM2free
Counter	FTM2	FTM2
⊟ Interrupt service/event	Enabled	
Interrupt	Vftm2fault_ovf	Vftm2fault_ovf
Interrupt priority	medium priority	level 4, priority withir
Interrupt period	500.0 ms	500 ms
Component uses entire timer	no	
⊟ Initialization		
Enabled in init. code	yes	
Events enabled in init.	yes	

You will find many components that are called embedded components in the Processor Expert Components library window. These components are designed to cover the most commonly required functionality used for the microcontroller applications, such as from handling port bit operations, external interrupts, and timer modes up to serial asynchronous/synchronous communications, A/D converter, I2C, and CAN.

By setting properties, you can define the behavior of the component in runtime. You can control properties in design time by using the Component Inspector. Runtime control of the component function is done by the methods. Events are interfacing hardware or software events invoked by the component to the user's code.

You can enable or disable the appearance (and availability) of methods of the component in generated source code. Disabling unused methods could make the generated code shorter. For more details, refer to the General Optimizations topic.

Events, if used, can be raised by interrupt from the hardware resource such as timer, SIO or by software reason, such as overflow in application runtime. You can enable or disable interrupts using component methods and define priority for event occurrence and for executing its Interrupt Service Routine (ISR). The hardware ISR provided by the component handles the reason for the interrupt. If the interrupt vector is shared by two (or more) resources, then this ISR provides the resource identification and you are notified by calling the user event handling code.

Creating Applications

Creation of an application with Processor Expert on any microcontroller is fast. To create an application, first choose and set up a processor component, add other components, modify their properties, define events and generate code. Processor Expert generates all code (well commented) from components according to your settings. For more details, refer to the Code Generation topic.

This is only part of the application code that was created by the Processor Expert processor knowledge system and solution bank. The solution bank is created from hand written and tested code optimized for efficiency. These solutions are selected and configured in the code generation process.

Enter your code for the events, provide main code, add existing source code and build the application using classic tools, such as compiler, assembler and debug it. These are the typical steps while working with Processor Expert.

Other components may help you to include pictures, files, sounds, and string lists in your application.

Processor Expert has built-in knowledge (internal definitions) about the entire microcontroller with all integrated peripherals. The microcontroller units and peripherals are encapsulated into configurable components called embedded components and the configuration is fast and easy using a graphical Component Inspector.

Peripheral Initialization components are a subset of embedded components that allow you to setup initialization of the particular on-chip device to any possible mode of operation. You can easily view all initialization values of the microcontroller produced by Processor Expert with highlighted differences between the last and current properties settings.

Processor Experts performs a design time checking of the settings of all components and report errors and warnings notifying you about wrong property values or conflicts in the settings with other components in the project. For more information, refer to the Design Time Checking: Consequences and Benefits topic.

Processor Expert contains many useful tools for exploring a structure of the target microcontroller showing the details about the allocated on-chip peripherals and pins.

Processor Expert generates a ready-to-use source code initializing all on-chip peripherals used by the component according to the component setup.

RTOS Support

Processor Expert provides a set of LDD components (Logical Device Drivers) that support generation of driver code that can be integrated with RTOSes (Real Time Operating Systems). For more details, refer to the Logical Device Drivers topic.

Terms and Definitions Used in Processor Expert

Component — An Embedded Component is a component that can be used in Processor Expert. Embedded Components encapsulate the functionality of basic elements of embedded systems like processor core, processor on-chip peripherals, standalone peripherals, virtual devices and pure software algorithms and wrap these facilities to properties, methods, and events (like objects in OOP). Components can support several languages (ANSI C, Assembly language or other) and the code is generated for the selected language.

Component Inspector — Window with all parameters of a selected component: properties, methods, events.

Bus clock — A main internal clock of the processor. Most of the processor timing is derived from this value.

Processor Component — Component that encapsulates the processor core initialization and control. This component also holds a group of settings related to the compilation and linking, such as Stack size, Memory mapping, linker settings. Only one processor component can be set active as the target processor. For details, refer to the Processor Components topic.

Component Driver — Component drivers are the core of Processor Expert code generation process. Processor Expert uses drivers to generate the source code modules for driving an internal or external peripheral according to the component settings. A Component can use one or more drivers.

Counter — Represents the whole timer with its internal counter.

Events — Used for processing events related to the component's function (errors, interrupts, buffer overflow etc.) by user-written code. For details, refer to the Embedded Components topic.

External user module — External source code attached to the PE project. The external user module may consist of two files: implementation and interface (`*.C` and `*.H`).

Free running device — Virtual device that represents a source of the overflow interrupt of the timer in the free running mode.

High level component — Component with the highest level of abstraction and usage comfort. An application built from these components can be easily ported to another microcontroller supported by the Processor Expert. They provide methods and events for runtime control. For details, refer to the Component Categories topic.

Internal peripherals — internal devices of the microcontroller such as ports, timers, A/D converters, etc. usually controlled by the processor core using special registers.

ISR - Interrupt Service Routine — code which is called when an interrupt occurs.

LDD components — Logical Device Driver components. The LDD components are efficient set of components that are ready to be used together with RTOS. They provide a unified hardware access across microcontrollers allowing to develop simpler and more portable RTOS drivers. For details, refer to the Component Categories topic.

Low level component — a component dependent on the peripheral structure to allow the user to benefit from the non-standard features of a peripheral. The level of portability is decreased because of this peripheral dependency. For details, refer to the Component Categories topic.

Microcontroller - Microcontroller Unit — microcontroller used in our application.

Methods — user callable functions or sub-routines. The user can select which of them will be generated and which not. Selected methods will be generated during the code generation process into the component modules.

Module - Source code module — could be generated by Processor Expert (Component modules, Processor Module, events.c) or created by the user and included in the project (user module).

OOP — Object-oriented programming (OOP) was invented to solve certain problems of modularity and reusability that occur when traditional programming languages such as C are used to write applications.

PE — Abbreviation of Processor Expert that is often used within this documentation.

PESL — *Processor Expert System Library* (PESL) is dedicated to power programmers, who are familiar with microcontroller architecture - each bit and each register. PESL provides the macros to access the peripherals directly, so PESL should be used only in some special cases. For details, refer to the Processor Expert System Library topic.

Peripheral Initialization component — encapsulates the whole initialization of the appropriate peripheral. Components that have the lowest levels of abstraction and usage comfort. For details, refer to the Component Categories topic. They usually do not support any methods or events except the initialization method. The rest of the device driver code needs to be written by hand using either PESL or direct control of the peripheral registers. For details, refer to the Low-level Access to Peripherals topic.

Popup menu — this menu is displayed when the right mouse button is pressed on some graphical object.

PLL — Phase Locked Loop. This circuit is often built-in inside the processor and can be used a main source of the clock within the processor.

Prescaler — A fixed or configurable device that allows to divide or multiply a clock signal for a peripheral processor peripheral or its part.

Properties — Parameters of the component. Property settings define which internal peripherals will be used by the component and also initialization and behavior of the component at runtime.

RTOS — Real Time Operating System is an operating system (OS) intended for real-time applications.

Processor — The processor derivative used in a given project.

Template — It is a component template with preset parameters.

User-defined Component Template — User-defined component template is a component with preset parameters saved under a selected name. Also the name of the author and short description can be added to the template.

User module — Source code module created or modified by the user. (Main module, event module or external user module).

Xtal — A crystal - a passive component used as a part of an oscillator circuit.

2
User Interface

The Processor Expert menu is integrated as a plugin in the Eclipse IDE providing set of views. The IDE main menu has a menu item named **Processor Expert**.

The user interface of Processor Expert consists of the following windows:

– Component Inspector — Allows you to setup components of the project.

– Component Library — Shows all supported components including processor components and component templates.

– Configuration Registers — Shows overview of the peripheral initialization settings for the current processor.

– Memory Map — Shows the processor address space and internal and external memory mapping.

– Components — Shows an embedded component that can be used in Processor Expert.

– Initialization Sequence — It is possible to customize the initialization sequence of components. By default, the sequence is not specified. You can change the sequence using up or down buttons. Initialization of processor component is always first.

– Processor — The processor derivative used in a given project.

This chapter explains:

- Main Menu
- Components View
- Components Library View
- Component Inspector View
- Processor View
- Memory Map View
- Configuration Registers View
- Initialization Sequence View

Main Menu

The Processor Expert plug-in is integrated into the Eclipse IDE as plugin application. The IDE main menu contains a new menu item named Processor Expert.

The Processor Expert menu includes:

- **Show views** — Shows standard Processor Expert windows in case they are hidden.
- **Hide views** — Hides Processor Expert views.
- **Import Components(s)** — This command allows to select and install Processor Expert update packages (`.PEUpd`) files. These files can be created in Component Development Environment (CDE) by exporting a user's component.

Project Pop-up Menu

This menu is available on right-clicking at the `ProcessorExpert.pe` file. It contains the standard commands with the Processor Expert specific command:

Generate Processor Expert Code — Invokes code generation for the current project. The generated files are automatically inserted into the active (default) target in the project. Generated files corresponding to the Embedded Components can be accessed from the **Generated_Code** folder. For more details, refer to the Code Generation topic.

For Processor Expert related settings and options, refer to the Processor Expert Options.

Processor Expert Options

This section contains the following topics:

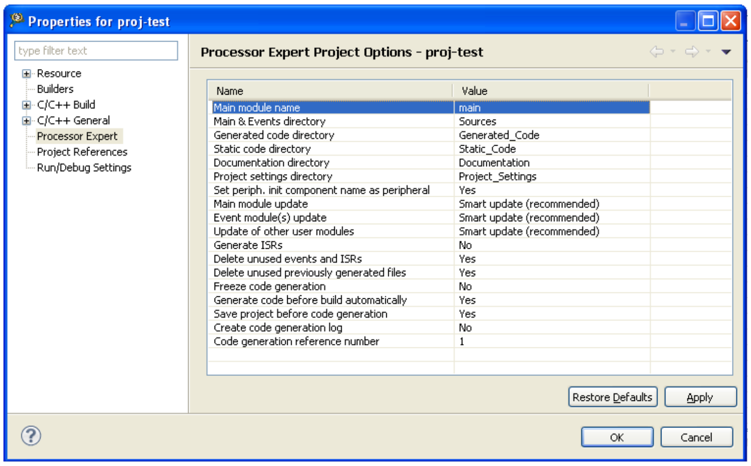
- Project Options
- Preferences

Project Options

Project options related to Processor Expert can be found in **Properties** dialog box. To access this dialog box, click **Project > Properties**. The **Properties** dialog box appears.

Select **Processor Expert** option in the list on the left. Description of the individual options can be found in the hint window displayed when the cursor is placed on an item.

Figure 2.1 Project Properties Dialog



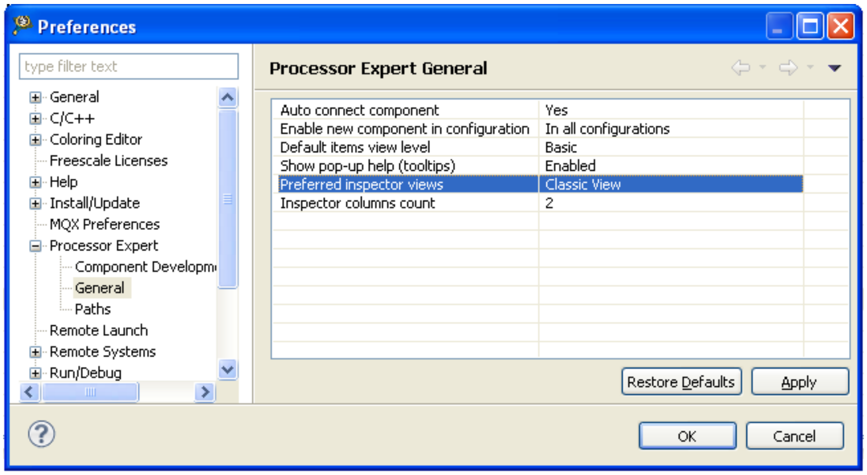
Name	Value
Main module name	main
Main & Events directory	Sources
Generated code directory	Generated_Code
Static code directory	Static_Code
Documentation directory	Documentation
Project settings directory	Project_Settings
Set periph. init component name as peripheral	Yes
Main module update	Smart update (recommended)
Event module(s) update	Smart update (recommended)
Update of other user modules	Smart update (recommended)
Generate ISRs	No
Delete unused events and ISRs	Yes
Delete unused previously generated files	Yes
Freeze code generation	No
Generate code before build automatically	Yes
Save project before code generation	Yes
Create code generation log	No
Code generation reference number	1

NOTE **Restore Defaults** button restore all settings to its default values except Code Generation reference number. This number indicates number of times code was generated for a given project.

Preferences

Global settings related to Processor Expert can be found in **Preferences** dialog available using the command **Window > Preferences**. The PE related items can be found under Processor Expert in the list on the left. Description of the individual options can be found in the hint window displayed when the cursor is placed on an item.

Figure 2.2 Preferences Dialog



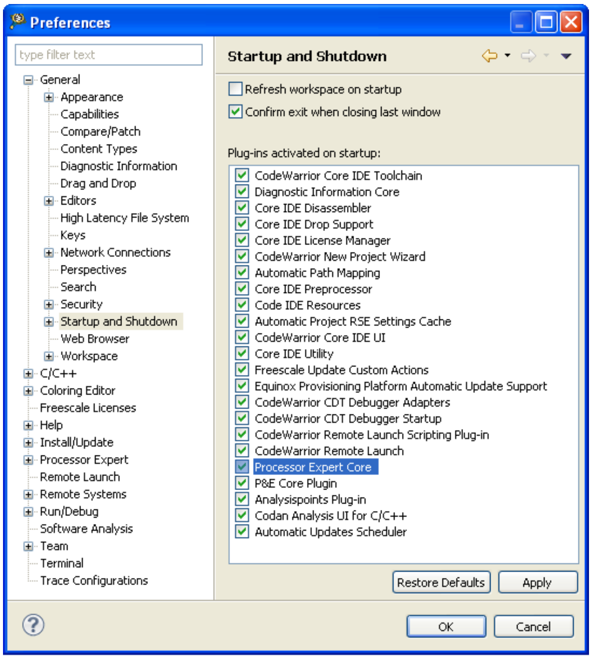
Auto connect component	Yes
Enable new component in configuration	In all configurations
Default items view level	Basic
Show pop-up help (tooltips)	Enabled
Preferred inspector views	Classic View
Inspector columns count	2

There is an option **Preferred inspector views** that allows you to decide how to display the tabs of **Component Inspector** view. The three views are **Custom**, **Classic**, and **Tabs**.

To start or shutdown the processor expert, click **Windows > Preferences** and expand **General** and select **Startup and Shutdown**.

Processor Expert starts after the Eclipse workbench user interface is displayed if the **Processor Expert Core** checkbox is selected as shown below.

Figure 2.3 Preferences — Startup and Shutdown



Startup and Shutdown

☐ Refresh workspace on startup
☑ Confirm exit when closing last window

Plug-ins activated on startup:

☑ CodeWarrior Core IDE Toolchain
☑ Diagnostic Information Core
☑ Core IDE Disassembler
☑ Core IDE Drop Support
☑ Core IDE License Manager
☑ CodeWarrior New Project Wizard
☑ Automatic Path Mapping
☑ Core IDE Preprocessor
☑ Code IDE Resources
☑ Automatic Project RSE Settings Cache
☑ CodeWarrior Core IDE UI
☑ Core IDE Utility
☑ Freescale Update Custom Actions
☑ Equinox Provisioning Platform Automatic Update Support
☑ CodeWarrior CDT Debugger Adapters
☑ CodeWarrior CDT Debugger Startup
☑ CodeWarrior Remote Launch Scripting Plug-in
☑ CodeWarrior Remote Launch
☑ Processor Expert Core
☑ P&E Core Plugin
☑ Analysispoints Plug-in
☑ Codan Analysis UI for C/C++
☑ Automatic Updates Scheduler

Components View

Components view shows the tree with the following items:

- **Generator_Configurations** — Configurations of the project.
- **Operating System** — contains special components that provide operating system interface and configuration if there are any used.
- **Processors** — contains Processor Components included in the project.
- **Components** — it is included in the project. Every component inserted in the project is displayed in the **Component Inspector** view and may have a sub tree showing items available for the component (note that components can offer only some or even none of these items):
  - **Methods** — Methods allow runtime control of the component's functionality.
  - **Events routines** — Events allow handling of the hardware or software events related to the component. If the event is disabled, the name of the event is shown. For enabled events, the name of the handling function is shown.

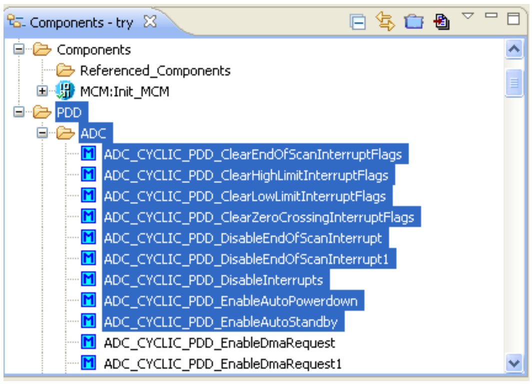
- **ISRs** — Represent component-related interrupt routines that is created by you for low-level interrupt processing. For items, whose ISR names have been specified within component settings, a user-specified name of an ISR and name of the interrupt vector is shown. If an ISR name is not specified (interrupt has to be disabled in this case), only the interrupt vector name is present.

  You can specify an ISR in the component and generate the code. If the **Generate ISRs** project option is selected, empty ISR templates are generated into an event module. If you disable an ISR and select the **Delete unused events and ISRs** project option, empty ISR templates are removed automatically.
- **PESL commands** — low-level PESL commands related to the peripheral configured by this component. This folder is available only for Peripheral Initialization components.
- **PDD Macros** — low-level PDD macros for peripherals allocated by the component. Macros can be dragged and dropped into the source code. If the PDD macro is dragged, the base address parameter is automatically filled with the macro-value which uses the peripheral selected in the component.

NOTE PESL and PDD folders are available only for Peripheral Initialization components and only if PDD library or PESL library is supported for the selected processor. Either PESL or PDD folder is displayed and not both.

- **PDD** — The global list of all PDD macros for all available peripherals are grouped by peripheral name. This folder is available only if PDD is available for the currently active CPU. PDD commands are low-level peripheral access macros and they are the replacement of PESL macros. PDD commands are available on all platforms supported by Logical Device Drivers (LDD). Macros can be dragged and dropped into the source code. For details on PDD, refer to the Low-level Access to Peripherals topic.

Figure 2.4 List of PDD Macros



Components
Referenced_Components
MCM:Init_MCM
PDD
ADC
ADC_CYCLIC_PDD_ClearEndOfScanInterruptFlags
ADC_CYCLIC_PDD_ClearHighLimitInterruptFlags
ADC_CYCLIC_PDD_ClearLowLimitInterruptFlags
ADC_CYCLIC_PDD_ClearZeroCrossingInterruptFlags
ADC_CYCLIC_PDD_DisableEndOfScanInterrupt
ADC_CYCLIC_PDD_DisableEndOfScanInterrupt1
ADC_CYCLIC_PDD_DisableInterrupts
ADC_CYCLIC_PDD_EnableAutoPowerdown
ADC_CYCLIC_PDD_EnableAutoStandby
ADC_CYCLIC_PDD_EnableDmaRequest
ADC_CYCLIC_PDD_EnableDmaRequest1

All component's items have status icons that signify the enabled or disabled state. If this state cannot be changed directly, the background of the icon is gray. For more details, refer to the Embedded Components topic.

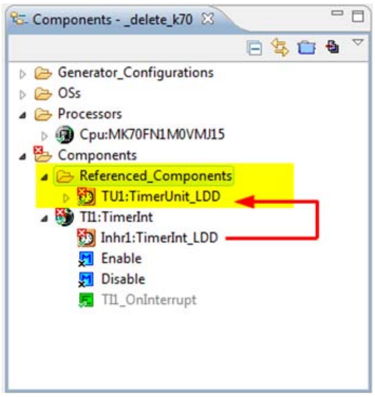
Shared components are automatically placed into a dedicated subfolder **Referenced_Components**. You can move the component from this folder to anywhere.

This table explains the various states of a component.

Table 2.1 Description of Component States

Component Status Icon	Description
✓	Signifies that component is enabled. It can be configured and code can be generated from this component.
✗	Signifies that component is disabled. It can be configured, but the configuration validation/ generation is disabled. No code is generated from this component.
🟥	Signifies error in the component. For example, Components folder contains component with error.
◆	Signifies that component is frozen and will not be re-generated. When the user generates the code again, files of this component are not modified and the generated code is frozen.

Figure 2.5 Referenced Components



When you have more than one Processor Expert project in your workspace and you are working with those projects, the last project shown in **Components** view is recorded in the workspace history. When you restart the Eclipse IDE, the last session project is opened automatically.

View Menu

- **Generate Processor Expert Code** — invokes code generation for the current project.
- **Synchronize Processor Expert Static Code** — forces static code synchronization. For details, refer to the Static Code Support in Processor Expert topic.
- **Close/Open Processor Expert Project** — closes the project if it is open or opens the project if it is closed.
- **Properties** — displays the Processor Expert properties for a specific project.
- **Edit Initialization Sequence** — modify the initialization sequence of components.
- **Export** — allows to export component settings or configuration of selected Processor Expert components.
- **Import** — allows to import component settings or configuration of selected Processor Expert components.

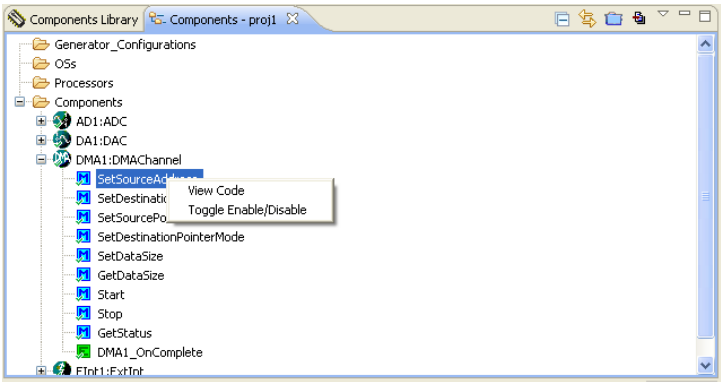
Pop-up Menu

- **Inspector** — opens **Component Inspector** view for the component. For more details, refer to the Component Inspector View topic.
- **Inspector - Pinned** — opens **Component Inspector** view for the component in "pinned" mode. This command allows to have several inspector views for different

components opened at once. For more details, refer to the Component Inspector View topic.

- **Code Generation** — allows to disable/enable the generated module for the component.
- **Configuration Registers** — displays the **Configuration Registers** view for the peripheral initialized by the selected component. For more details, refer to the Configuration Registers View topic.
- **Target Processor Package** — displays the **Processor** view for the processor derivative used in a given project.
- **Processor Memory Map** — displays the **Memory Map** view for the processor address space and internal and external memory mapping.
- **Rename Component** — changes the name of the component.
- **Select Distinct/Shared mode** — switches between shared and distinct mode of the component. This setting is available for LDD components only. For more details, refer to the Logical Device Drivers topic.
- **Open File** — opens the generated code from the selected component for the source code editor. Note that the code is available only after successful code generation.
- **Component Enabled** — enables/disables component in the project.
- **Remove component from project** — deletes the component from the project.
- **Help on component** — shows a help page for the component.
- **Save Component Settings As Template** — creates a template of the selected component. For more details, refer to the Creating User Component Templates topic.
- **View Code** — Opens code editor at the code of the selected method or event.
- **Toggle Enable/Disable** — Enables/Disables the Method or Event.

Figure 2.6 Components View



Components Library View

The **Components Library** view shows supported embedded components including processor components and component templates. It lets you select a desired component or template and add it to the project.

Modes

The Components Library has the following four tabs allowing you to select components in different modes:

- **Categories** — Contains all available components. The components are sorted in a tree based on the categories defined in the components. For more details, refer to the Component Categories topic.
- **Alphabetical** — Shows alphabetical list of the available components.
- **Assistant** — Guides you during the component selection process. The user answers a series of questions that finally lead to a selection of a component that suits best for a required function. For more details, refer to the Component Assistant topic.
- **Processors** — Contains list of the available processors.

Component names are colored black and the component template names are colored blue. The components that are not supported for the currently selected target processor are gray. By double-clicking on the component, it is possible to insert the component into the current project. The description of the component is shown in a hint.

Filtering

Filter can be activated using the filtering icon. If it is active, only the components that could be used with the currently selected target processors are shown.

If the filter is inactive, Processor Expert also shows components that are not available for the current processor.

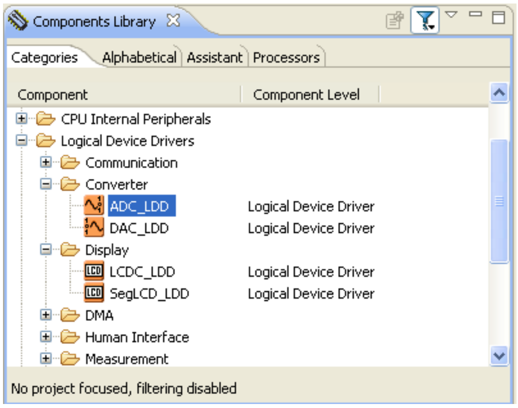
Pop-up Menu

A pop-up menu opens by right-clicking a component or folder. It contains the following commands:

- **Add to project** — Adds the component to the current project.
- **Add to project with wizard** — Adds the component to the current project and opens a configuration wizard.
- **Expand all** — Expands the folder and all its subfolders.
- **Collapse all** — Collapses the folder and all its subfolders.

- **Refresh** — Refreshes the view area.
- **Delete** — Only user templates and components are deleted. User component is deleted from the folder `<Processor Expert Install>/ProcessorExpert/Beans/<ComponentToBeDeleted>`. Other files like `*.inc`, `*.drv`, `*.src` remain intact.
- **Help on component** — Opens help information for the selected component.

Figure 2.7 Components Library



- CPU Internal Peripherals
- Logical Device Drivers
  - Communication
  - Converter
    - ADC_LDD — Logical Device Driver
    - DAC_LDD — Logical Device Driver
  - Display
    - LCDC_LDD — Logical Device Driver
    - SegLCD_LDD — Logical Device Driver
  - DMA
  - Human Interface
  - Measurement

No project focused, filtering disabled

Component Assistant

The Component Assistant is a mode of **Components Library** view. It guides you during the selection of components, that is basic application building blocks. You will have to answer a series of questions that finally lead to a selection of a component that suits best for a required function. In this mode, the **Components Library** view consists of the following parts:

- History navigation buttons and the history line showing answers for already answered questions. You can walk through the history using the arrow buttons or by clicking the individual items.
- A box with a current question.
- A list of available answers for the current question.

  If the answer already corresponds to a single component (it has an icon of the component and there is a [component name] at the end of the list line) and user double-clicks it, it is added into the project. Also, you can right-click on the line to open the pop-up menu of the component, allowing to add it into the project or view its documentation (for details, refer to the Components Library View topic).

If more questions are necessary for the component selection, the line with the answer contains a group icon and in brackets a number of components that still can possibly be selected. After clicking on such line a next question is displayed.

This mode of Components Library does not offer addition of processor components. If you would like to add another processor component, switch to processors tab.

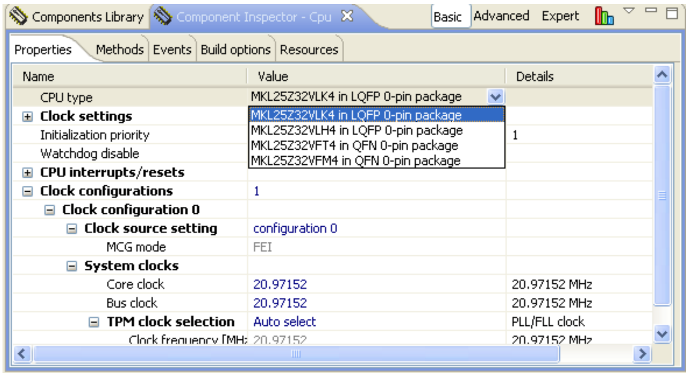
Component Inspector View

Component Inspector allows to view and edit attributes of the item selected in the Project Explorer.

Inspector window contains three columns:

- **Name** — Name of the item to be selected. Groups of items may be collapsed or expanded by double clicking on the first line of the group with its name, it has '+' or '-' sign on the left.
- **Value** — the settings of the items are made in this column. For list of item types, refer to the Inspector Items topic for details.
- **Details** — the current setting or an error status may be reflected on the same line, in the rightmost column of the inspector.

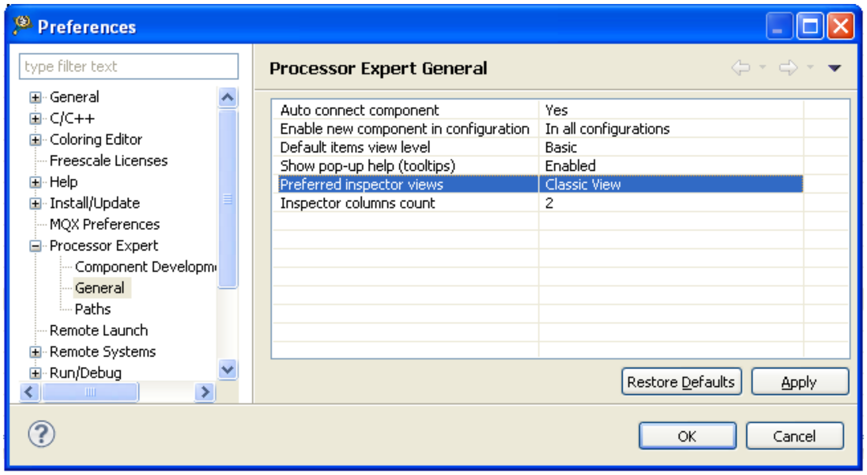
Figure 2.8 Component Inspector View — Displaying Pin variant and Package



Name	Value	Details
CPU type	MKL25Z32VLK4 in LQFP 0-pin package	
⊞ Clock settings	MKL25Z32VLK4 in LQFP 0-pin package	
Initialization priority	MKL25Z32VLH4 in LQFP 0-pin package	1
Watchdog disable	MKL25Z32VFT4 in QFN 0-pin package	
⊞ CPU interrupts/resets	MKL25Z32VFM4 in QFN 0-pin package	
⊟ Clock configurations	1	
⊟ Clock configuration 0		
⊟ Clock source setting	configuration 0	
MCG mode	FEI	
⊟ System clocks		
Core clock	20.97152	20.97152 MHz
Bus clock	20.97152	20.97152 MHz
⊟ TPM clock selection	Auto select	PLL/FLL clock
Clock frequency [MHz	20.97152	20.97152 MHz

The Component Inspector can be displayed in three views. The views can be selected from the **Preferences** dialog. On the left side of the **Preferences** dialog, select **Processor Expert > General**. On the right side, select **Preferred inspector views** from the drop-down list as shown below.

Figure 2.9 Preferred inspector views



Auto connect component	Yes
Enable new component in configuration	In all configurations
Default items view level	Basic
Show pop-up help (tooltips)	Enabled
Preferred inspector views	Classic View
Inspector columns count	2

The three views are:

- **Classic** — this is the default tree-based view as shown in the Figure 2.9. In default view, built-in tabs are displayed in the sequence "Properties, methods, events," and the tabs that are specific to custom view are added at the end. In case, **Classic** view option is selected, the sequence of tabs for component XYZ will be:

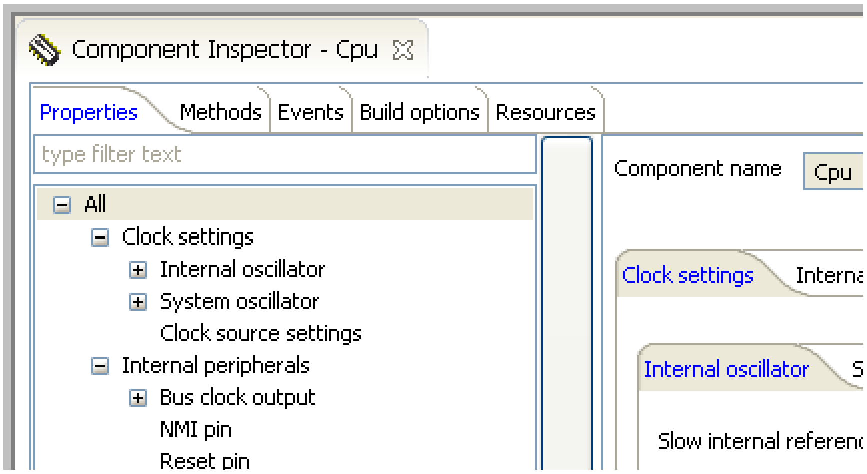
  Properties | Methods | Events | Device Settings

- **Custom** — when the **Custom** view is selected, it is marked as replacement of built-in tabs and replaces their position with the built-in tab. In case, **Custom** view option is selected, the sequence of tabs for component XYZ will be:

  Device Settings | Methods | Events | Properties

- **Tabs** — this view displays configuration elements in graphical form, better organized, easy to understand and use. Nested tabs and tables are used to present complex configurations as shown in figure below.

Figure 2.10 Tabs View

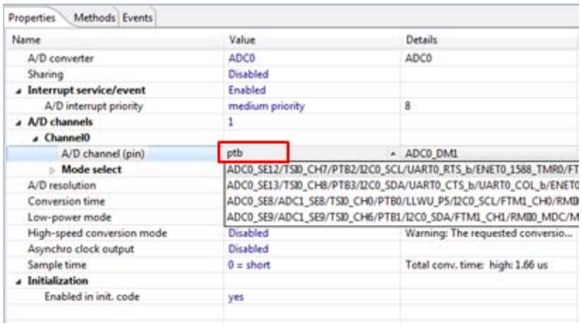


NOTE After selecting the required view, restart or switch the workspace to display the selected view in Component Inspector.

In the **Preferences** dialog, you can also choose the number of columns that will appear in **Component Inspector** view for displaying properties. Select **Inspector columns count** and choose the value from 1 to 4 from the drop-down list. This option is valid only for **Tabs** view.

You can filter pin names in the **Component Inspector** view. It is now easy to select pin from the long drop- down list. You can type the text and filter list of available values. The filtering is enabled only when more than 3 values are available in the list.

Figure 2.11 Filtering Pin Names



Name	Value	Details
A/D converter	ADC0	ADC0
Sharing	Disabled	
Interrupt service/event	Enabled	
A/D interrupt priority	medium priority	8
A/D channels	1	
Channel0		
A/D channel (pin)	ptb	ADC0_DM1
Mode select		
A/D resolution		
Conversion time		
Low-power mode		
High-speed conversion mode	Disabled	Warning: The requested conversio...
Asynchro clock output	Disabled	
Sample time	0 = short	Total conv. time: high: 1.66 us
Initialization		
Enabled in init. code	yes	

ADC0_SE12/TSI0_CH7/PTB2/I2C0_SCL/UART0_RTS_b/ENET0_1588_TMR0/FT
ADC0_SE13/TSI0_CH8/PTB3/I2C0_SDA/UART0_CTS_b/UART0_COL_b/ENET0
ADC0_SE8/ADC1_SE8/TSI0_CH0/PTB0/LLWU_P5/I2C0_SCL/FTM1_CH0/RMII
ADC0_SE9/ADC1_SE9/TSI0_CH6/PTB1/I2C0_SDA/FTM1_CH1/RMII0_MDC/M

Read Only Items

Some items are read-only, you can not change the value and are displayed in gray.

View Mode Buttons

They are placed at the top of the window (Basic, Advanced). These buttons allow you to switch complexity of the view of the component's items. Refer to the Items Visibility topic for details.

View Menu

This menu can be invoked by clicking on the down arrow icon. The menu contains the following commands:

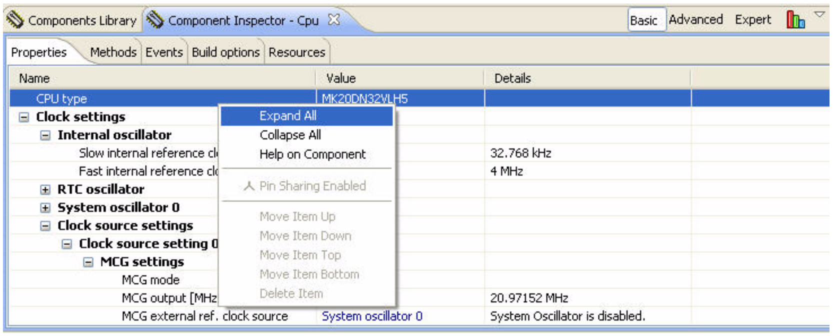
- **Basic, Advanced** — view mode switching. These options have the same meaning as the view mode buttons.
- **Ignore Constraints and non-Critical Errors** — this option enables a special mode of Processor Expert. In this mode, Processor Expert allows you to generate output files, even though some settings may break some constraints or limits and errors are reported.
- **Expand All** — if a group is selected, expands all items within the selected group. Otherwise, all groups in the Inspector are expanded. If the expanded group contains any groups that are disabled (gray), the user is asked if the disabled groups should all be expanded.
- **Collapse All** — if a group is selected, collapses all items within the selected group. Otherwise, all groups in the Inspector are collapsed.
- **Help on Component** — shows a help page for the component.
- **Save component settings as template** — creates a template for the current component settings. Refer to the Creating User Component Templates topic for details.
- **Edit comment** — when comment is empty, icon appears gray. If you have entered the comment, icon appears yellow on the toolbar. The comment is displayed in tooltip for the current component or configuration. For example, you can enter the note for the CPU component, such as "Please do not enter value more than 25MHz in Clock settings." These comments are added in the generated source file, `Cpu.h` file.
- **Open New pinned view** — opens a copy of the inspector for currently selected component. This command allows to have several inspector views for different components opened at once.

- **Tabs view** — this view displays configuration elements in graphical form, better organized, easy to understand and use. Nested tabs and tables are used to present complex configurations.
- **Search** — searches Inspector item by name. It also accepts wild cards like * or ? (* =any string and ? = any character).

Pop-up Menu

This menu is invoked by right-clicking a specific inspector item. The menu contains the following commands:

Figure 2.12 Component Inspector View — Pop-up Menu



Name	Value	Details
CPU type	MK20DN32VLH5	
Clock settings		
Internal oscillator		
Slow internal reference cl		32.768 kHz
Fast internal reference cl		4 MHz
RTC oscillator		
System oscillator 0		
Clock source settings		
Clock source setting 0		
MCG settings		
MCG mode		
MCG output [MHz		20.97152 MHz
MCG external ref. clock source	System oscillator 0	System Oscillator is disabled.

- **Expand All** — if a group is selected, expands all items within the selected group. Otherwise, all groups in the inspector are expanded. If the expanded group contains any groups that are disabled (gray), the user is asked if the disabled groups should all be expanded.
- **Collapse All** — if a group is selected, collapses all items within the selected group. Otherwise, all groups in the inspector are collapsed.
- **Help on Component** — shows a help page for the component.
- **Pin Sharing Enabled** — enables the pin sharing. This command is available only for pin properties. For more information, refer to the Pin Sharing topic.
- **Move Item Up** — moves the item up in the list.
- **Move Item Down** — moves the item down in the list.
- **Move Item Top** — moves the item on the top of the list.
- **Move Item Bottom** — moves the item at the bottom of the list.

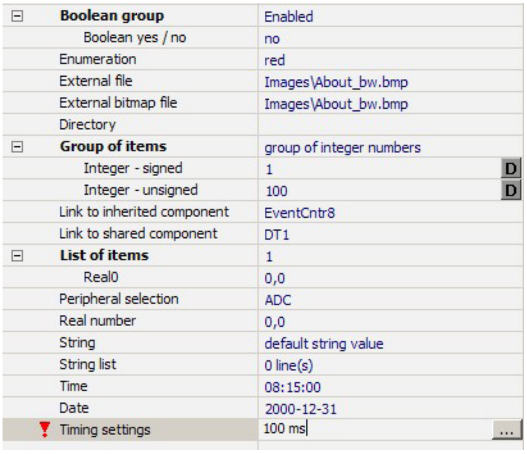
NOTE **Move** options are enabled for ListItemFromFIle property.

- **Delete Item** — does not delete the component, but can delete the property item from the list of property. The list of items can have some constraints on minimal or maximum number of items. Add ADC component into the project and add at least one extra channel then you will be able to see this option enabled.

Inspector Items

The following types of the items are there in the **Component Inspector** view.

Figure 2.13 Example Component with Various Inspector Item Types



Boolean group	Enabled
Boolean yes / no	no
Enumeration	red
External file	Images\About_bw.bmp
External bitmap file	Images\About_bw.bmp
Directory	
Group of items	group of integer numbers
Integer - signed	1
Integer - unsigned	100
Link to inherited component	EventCntr8
Link to shared component	DT1
List of items	1
Real0	0,0
Peripheral selection	ADC
Real number	0,0
String	default string value
String list	0 line(s)
Time	08:15:00
Date	2000-12-31
Timing settings	100 ms

Table 2.2 explains the various types of items.

Table 2.2 Inspector Item Types

Field	Description
Boolean Group	A group of settings controlled by this boolean property. If the group is enabled, all the items under the group are valid; if it is disabled, the list of items is not valid. Clicking + sign will show/hide the items in the group but doesn't affect value or validity of the items.
Boolean yes/no	You can switch between two states of the property using a drop-down menu. The Generate code/Don't generate code settings of methods and events works the same way and determines whether the implementation code for the corresponding method or event will be generated or not (you may thus generate only the methods and events used by your application).
Enumeration	Selection from a list of values. If you click the arrow icon ( ), a list of the possible values for the property is offered. For some derivatives, pin and package details are displayed for processor variant.
Enumeration Group	A list of items. Number of visible (and valid) items in the group depends on chosen value. Clicking the arrow icon ( ) will show a list of the possible values of the property. Clicking the + sign shows/hides the items in the group but does not influence value or validity of the items.
File/Directory Selection	Allows to specify a file or directory. Clicking the icon opens a system dialog window allowing to select a file/directory.
Group	A list of items that can be expanded/ collapsed by clicking on the plus/minus icon or by double-clicking at the row. Values of the items in the group are untouched.

Table 2.2 Inspector Item Types

Field	Description
Integer Number	You can insert a number of a selected radix. Radix of the number could be switched using the icons (D = Decimal, H = Hexadecimal, B = Binary). Only reasonable radixes are offered for the property. If the radix switching icon is not present, Processor Expert expects the decimal radix.
Link to Inherited component	The down-arrow button allows to change the ancestor from the list of possible ancestor. Refer to the Component Inheritance and Component Sharing topic for details.
Link to shared component	The down-arrow button allows to change the component from the list of the available components or add a new component to the project. Refer to the Component Inheritance and Component Sharing topic for details.
List of items	A list of items may be expanded/collapsed by clicking on the plus/minus button in the left side of the row or by double clicking on the row. You may add/remove items by clicking on the plus/minus button. The items in the list can be arranged using the Pop-up Menu related commands.
Peripheral selection	You can select a peripheral from the list of the available peripherals. The peripheral that are already allocated have the component icon in the list. The properties that conflicts with the component settings have the red exclamation mark.
Real Number	You can insert any real (floating point) number.

Table 2.2 Inspector Item Types

Field	Description
String	Allows to enter any text or value. If there are no recommended values specified, there is no change in the user interface. If the values are specified, these values helps user to select typical or recommended value for the property. This feature is supported for String and Real Number properties. The recommended values used for the property are ONE, TWO, and THREE. The string property can offer predefined values accessible hitting key stroke `ctrl+space`. You can see that this is available when the string value is edited and there are predefined values available, small yellow bulb is displayed before top-left corner of edit field.
String list	Clicking the browse button (...) opens the simple text editor that allows to enter an array of text lines.
Time, Date	Allows to setup the Time/Date in a format according to the operating system settings.
Timing settings	Allows a comfortable setting of the component's timing. The timing dialog box opens on clicking on browse button (...). Refer to the Dialog Box for Timing Settings topic for details.

Items Visibility

Processor Expert supports **selectable visibility** of the component items. Each item is assigned a predefined level of visibility. **Higher visibility level** means that items with this level are more special and rarely used than the others with the lower visibility level. Component Inspector displays only items on and below the selected level. It could help especially beginners to set only basic properties at first and do optimization and improvements using advanced and expert properties or events later. There are three visibility levels:

- **Basic view** — The key and most often used items that configure the basic functionality of the components.

- **Advanced view** — All items from **Basic** view and the settings that configure some of more advanced and complex features of the component.

The visibility can be switched in the Component Inspector using **Basic** and **Advanced** buttons or within its view menu.

NOTE If an error occurs in a property with a higher visibility level than the level currently selected, then also this error is displayed.

Pin Settings

New pin model in Processor Expert is currently supported for Vybrid derivative. It allows to specify requirements for the pin configuration from different components. By default, property for pin selection contains **Automatic** value which represents no requirement for the configuration. If there is no requirement from the property:

- If there are requirement for pin assignment from another component, this pin is used
- If there are no requirements from any component, default assigned pin is used
- If there is no default assigned pin or the default assigned pin is in conflict with another configuration, no pin is assigned to the property

The requirements for pin configuration is specified from several components. Duplicated requirements does not represent any conflict and are accepted.

PinSettings component allows to specify pin configuration for all pins of the processor. You have the choice to specify pin routing either in PinSettings component or directly in LDD components (or both).

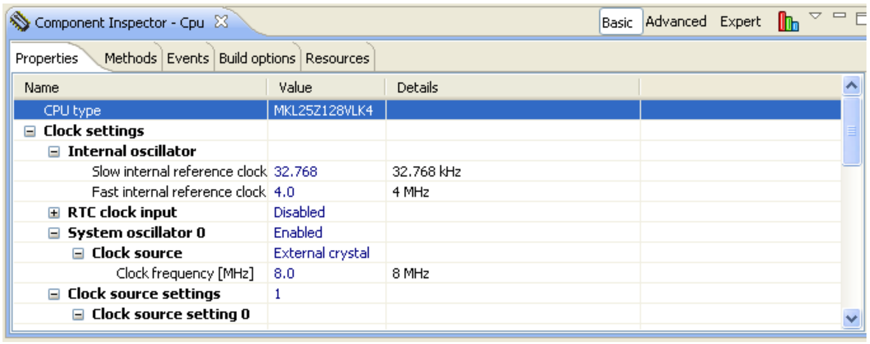
For automotive applications, it is expected that HW designer specify pin (pin routing) during board design (HW perspective is used) and sends it to SW engineers as an input for configuration of the SW application.

Component Inspector

Component Inspector is one of the inspector view variants intended to configure component settings. It allows to setup Properties, Methods, and Events of a component. Use command **Help on Component** from **View** menu (invoked by the down arrow icon) to see documentation for currently opened component.

NOTE Property settings influencing the hardware can often be better presented by the processor package view using the **Processor** view. Refer to the Processor View topic for details.

Figure 2.14 Component Inspector View



Name	Value	Details
CPU type	MKL25Z128VLK4	
⊟ **Clock settings**		
⊟ **Internal oscillator**		
Slow internal reference clock	32.768	32.768 kHz
Fast internal reference clock	4.0	4 MHz
⊞ **RTC clock input**	Disabled	
⊟ **System oscillator 0**	Enabled	
⊟ **Clock source**	External crystal	
Clock frequency [MHz]	8.0	8 MHz
⊟ **Clock source settings**	1	
⊟ **Clock source setting 0**		

The **Build options** page is present only in the processor component and it provides access to the settings related to linker and compiler.

The **Resources** page shows list of the processor component resources. You can also manually block individual resources for using them in Processor Expert.

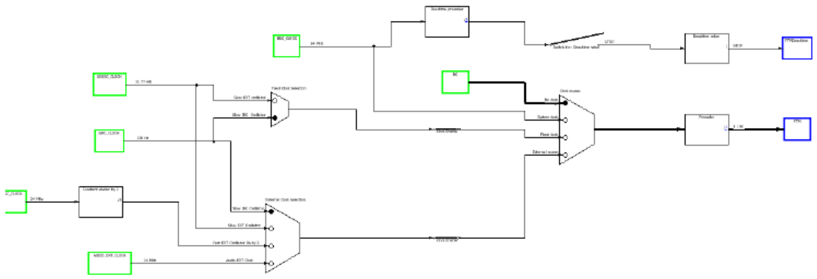
The page consists of the three columns:

- First shows the name of the resource. Resources are in groups according to which device they belong to.
- Second column allows you to reserve resource (for example pin) for external module. Click on icon to reserve/free a resource. Reserved resource is not used in Processor expert any more.
- Third column shows the current status of the resource and the name of the component which uses it (if the resource is already used).

For more details on component inspector items, refer to the Dialog Box for Timing Settings and Syntax for the Timing Setup in the Component Inspector topic.

In the **Component Inspector** view, you can view the clock diagram for only vybrid derivative. Create a project with Vybrid `MVF50GS10xx` derivative and add some peripheral initialization components, for example, `Init_FTM`. See the **Clock Diagram** tab in the **Inspector** view.

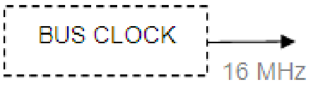
Figure 2.15 Inspector View — Clock Diagram



The following shows the graphical schematic of each element:

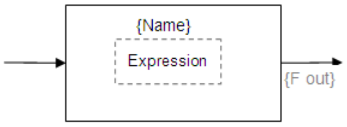
- Clock Source — It represents XTAL, internal oscillator, external pin or signal from another peripheral.

Figure 2.16 Bus Clock



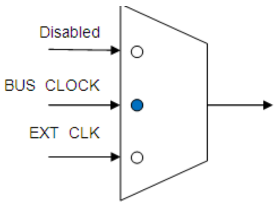
- Expressions — This element represents general expression, defined by element name and expression function, followed by output frequency.

Figure 2.17 Expression



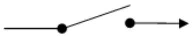
- Clock Selection — It represents selection of clock signal, optionally input signals may represent disabled clock.

Figure 2.18 Clock Selection



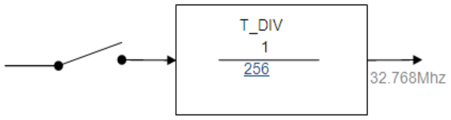
- Switch — It represents disabled path (optional).

Figure 2.19 Switch



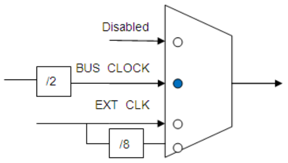
- More complex elements should be interpreted as combination of all the above elements. For example, clock selection with prescaler. It represents selection of clock source and divider by the single bit-field. There are three sub-variants with different graphical representation:
  - Divider that can be disabled optionally.

Figure 2.20 Clock Selection with Prescaler



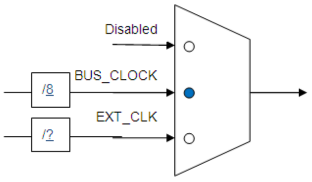
  - Upto 8 clock sources and some of them with divider.

Figure 2.21 Multiple Clock Sources



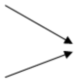
  - Different clock sources with high number of dividers.

Figure 2.22 Clock Sources with Dividers



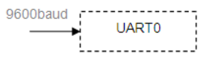
- Clock Branch — Branch is represented by connection of several elements to one source element.

Figure 2.23 Clock Branch



- Clock consumer or peripheral

Figure 2.24 Clock Consumer



Dialog Box for Timing Settings

The Timing dialog box provides a user-friendly interface for the settings of the component timing features. When you click the ... button of a timing item in the **Component Inspector** view, the Timing dialog box is displayed.

Before you start to edit component timing, you should set:

- Target processor in the **Components** view
- Used peripherals in the component's properties
- Supported speed modes in the component's properties

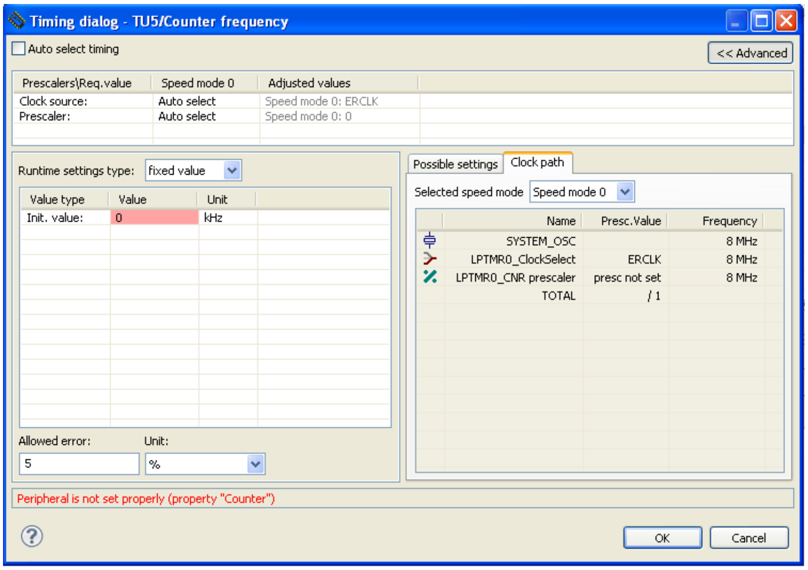
The settings are instantly validated according to the Processor Expert timing model. For details on the timing settings principles, refer to the Timing Settings topic.

Timing Dialog Box Controls

Timing dialog allows to select clock source manually. To access clock source, click **Advanced** button. You can manually select the value for prescaler and clock source.

Clock path shows current clock path from source to consumer, including all used prescalers and its configuration. Additionally, it shows frequency at each point of path.

Figure 2.25 Timing Settings Dialog Box



Prescalers\Req.value	Speed mode 0	Adjusted values
Clock source:	Auto select	Speed mode 0: ERCLK
Prescaler:	Auto select	Speed mode 0: 0

Runtime settings type: fixed value

Value type	Value	Unit
Init. value:	0	kHz

Allowed error: 5 Unit: %

Selected speed mode: Speed mode 0

Name	Presc.Value	Frequency
SYSTEM_OSC		8 MHz
LPTMR0_ClockSelect	ERCLK	8 MHz
LPTMR0_CNR prescaler	presc not set	8 MHz
TOTAL	/ 1	

Peripheral is not set properly (property "Counter")

Auto select timing option

This option is not supported for all components. It is supported only for timer, where the requirement can be specified both for counter and for compare/capture/overrun on this counter. For example, if Auto select timing option is selected for counter, this timing is configured based on peripherals for compare/capture/overrun. And if this option is selected for overrun, it is configured based on requirement for counter configuration. Currently, it is supported only in `timerUnit_LDD` component.

Runtime Setting Configuration

NOTE Runtime setting cannot be selected in the Basic view mode.

Runtime setting determines how the timing setting can be modified at runtime. The following options available are:

- **fixed value** — Timing cannot be changed at runtime.
- **from a list of values** — Allows to change the timing by selecting one of predefined values (from the list) using component method "SetXXXMode". This method sets the value(s) corresponding to the selected timing into the appropriate prescaler and other peripheral register(s). The values (modes) in the list can be added/removed by editing the timing values table.

- **from interval** — Allows to change a timing freely within a selected interval, while all values of the interval are selected with specified precision. Prescaler value is fixed in this mode, timing is set only using compare/reload registers value. It means that it is possible to reach all values within the interval by using the same prescaler.

Note that this kind of runtime setting requires runtime computations that can be time and space consuming and may not be supported on all microcontrollers.

NOTE Some of the methods used for runtime setting of timing will be enabled only if the appropriate runtime setting type is selected.

Timing Values Table

This table allows to set or modify a requested value(s) for the configured timing. Each row represents one time value and the number of rows depends on the selected type of runtime setting.

- For the option **fixed value**, there is only one row (Init.Value) containing the fixed initialization value.
- For the option **from a list of values**, there is one row for each of the possible timing modes. It is possible to enter 16 possible values (modes). The empty fields are ignored. You can drag and drop rows within the table to change their order. Refer to the Runtime Setting Configuration topic for more information.
- For the option **from interval**, the table has three rows that contain the Initial value, low limit and high limit of the interval. Refer to the Runtime Setting Configuration topic for details on this type of runtime setting.

There are two editable columns:

- **Value** — Fill in a requested time value (without units). The drop-down arrow button displays a list of values and you can select one of them. It is also possible to set the value by double-clicking on a value from the settings table.
- **Units** — Time units for the value. Refer to the Syntax for the Timing Setup in the Component Inspector topic for details.

Timing Precision Configuration

It is possible to specify desired precision of the timer settings by using one of the following settings (which one is used depends on the type of the timing):

- The field **Allowed error** allows to specify a tolerated difference between the real timing and the requested value. The **Unit** field allows to specify the units for the error allowed field (time units or a percentage of the requested value).
- The field **Min. resolution** is used for setting interval or capture component timing. Allows you to specify maximum length of one tick of the timer.

In case of interval settings type, the **% of low limit** (percentage of the low limit value) can be used as the unit for this value.

Minimal Timer Ticks

NOTE This item is available only for setting of period in components where it is meaningful, for example PWM, PPG.

It represents requirement for minimal number of timer ticks for specified period (usually it affects minimal value set into reload or modulo register). This is useful for configurations where it is expected to change period or duty in runtime, and in this case the parameter affects supported scale for such changes. There will be guarateed that there will be at least the given number of distinct values available for adjusting the duty of output signal. This will also be guaranteed for any available period of the signal.

Adjusted Values

This table displays real values for each speed mode of the selected row in the Timing values table.

These values are computed from the chosen on-chip peripheral settings, selected prescaler(s) value and the difference between a value selected by the user and the real value.

Status Box

The status box displays a status of the timing setting(s). If the timing requirements are impossible to meet, a red error message is displayed, otherwise it is blank and gray.

Possible Settings Table

This table is displayed on the right side of the timing dialog box when you click the **Possible settings** button on the top. The table displays values supported by the Processor for the selected peripheral.

If there are only individual values available to set, the table contains a list of values, each row represents one value. If there are intervals with a constant step available, each row contains one of the intervals with three values: **From**, **Till** - minimum and maximum value, **Step** - a step between values within the interval.

The way the values are displayed may be dependent on:

- **Runtime setting type** — If it is **fixed value** or **from list of values** the values present in rows (overlapping intervals) are shown only once. If **from time interval** runtime setting type is used, all intervals that can be set by various prescalers combinations are shown, even if they overlap. It is because intervals can differ in resolution, that is number of individual timing steps that can be achieved within them.

- **Timing unit** — If a frequency unit is used (for example, Hz, kHz), the step column is not visible.

By clicking on the table header, there is possible to order the rows as per selected column. By clicking the same column again, you can arrange the rows in ascending or descending order.

Double-clicking on a value will place the value into the currently edited row within the Timing values table.

The values listed in the possible settings table depend on the following timing settings:

- prescalers
- minimal timer ticks

and it also depends on

- selected processor
- selected peripheral
- speed-modes enabled for the component

The table contains a **speed mode** tabs (speed modes and related settings are supported only in the **Expert** view mode) that allow to filter the displayed intervals for a specific speed mode or show intersection of all. Note that the intersection contains only values that can be set in all speed modes with absolute precision (without any error), so some values that are still valid, but due to non-zero Error allowed, values are not shown.

Syntax for the Timing Setup in the Component Inspector

The properties that contain timing settings can be configured using the timing dialog box (For details, refer to the Dialog Box for Timing Settings topic) or directly entering the timing value. If the timing values are specified directly, it is necessary to enter not only a value (integer or real number) but also the unit of that value. The following units are supported:

- **microseconds** — A value must be followed by `us`.
- **milliseconds** — A value must be followed by `ms`.
- **seconds** — A value must be followed by `s`.
- **Processor ticks** — A unit derived from the frequency of external clock source. If there is no external clock enabled or available, it is derived from the value of internal clock source. A value must be followed by `ticks`.
- **Timer ticks** — A unit representing number of changes (for example increments) of the counter used by the component. The real time of one tick is influenced by input clock set for the timer.

- **Hertz** — A value must be followed by `Hz`.
- **kilohertz** — A value must be followed by `kHz`.
- **megahertz** — A value must be followed by `MHz`.
- **bit/second** — A value must be followed by `bits`.
- **kbit/second** — A value must be followed by `kbits`.

For example, if you want to specify 100 milliseconds, enter `100 ms`.

For more details on timing configuration, refer to the Timing Settings topic.

Configuration Inspector

Configuration Inspector is a variant of an inspector window. It shows the settings that belong to selected component. It could be invoked from configurations pop-up menu in the **Components** view (click on a configuration with the right-button and choose the Configuration Inspector). For details on configurations, refer to the Configurations topic.

Properties

The **Properties** tab contains optimization settings related to the configuration. These settings should be used when the code is already debugged. It could increase speed of the code, but the generated code is less protected for the unexpected situations and finding errors could be more difficult.

Note that some of the options may not be present for all Processor Expert versions.

- **Ignore range checking** — This option can disable generation of the code that provides testing for parameter range. If the option is set to `yes`, methods do not return error code `ERR_VALUE` neither `ERR_RANGE`. If the method is called with incorrect parameter, it may not work correctly.
- **Ignore enable test** — This option can disable generation of the code that provides testing if the component/peripheral is internally enabled or not. If the option is set to `yes`, methods do not return error code `ERR_DISABLED` neither `ERR_ENABLED`. If the method is called in unsupported mode, it may not work correctly.
- **Ignore speed mode test** — This option can disable generation of the code, that provides a testing, if the component is internally supported in the selected speed mode. If the option is set to `yes`, methods do not return error code `ERR_SPEED`. If the method is called in the speed mode when the component is not supported, it may not work correctly.
- **Use after reset values** — This option allows Processor Expert to use the values of peripheral registers which are declared by a chip manufacturer as default after reset values. If the option is set to `no`, all registers are initialized by a generated code, even

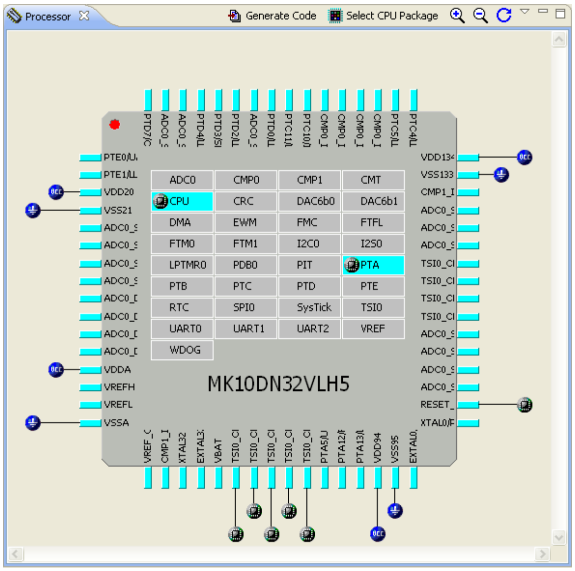
if the value after reset is the same as the required initialization value. If the option is set to `yes`, the register values same as the after reset values are not initialized.

- **Complete initialization in Peripheral Init. Component** — This option can disable shared initialization peripheral in Init methods of Peripheral Initialization Components. If this option is set to `yes`, the complete peripheral initialization is provided in Init method, even for parts that are already initialized in processor or elsewhere. It could mean longer code, but the initialization can be repeated in application using the Init method.

Processor View

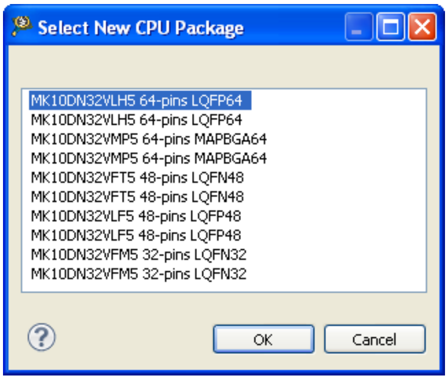
This view displays selected target microcontroller with its peripherals and pins. It allows you to generate code from processor and also to switch the CPU package. To open this vew, click **Window > Show View > Other...** and select **Processor Expert > Processor**.

Figure 2.26 Processor View



You can change the CPU package when the **Components** view is not being displayed by selecting the **Select New CPU Package** option on **Processor** view.

Figure 2.27 Processor View — Select New CPU Package



MK10DN32VLH5 64-pins LQFP64
MK10DN32VLH5 64-pins LQFP64
MK10DN32VMP5 64-pins MAPBGA64
MK10DN32VMP5 64-pins MAPBGA64
MK10DN32VFT5 48-pins LQFN48
MK10DN32VFT5 48-pins LQFN48
MK10DN32VLF5 48-pins LQFP48
MK10DN32VLF5 48-pins LQFP48
MK10DN32VFM5 32-pins LQFN32
MK10DN32VFM5 32-pins LQFN32

Control Buttons

The following table lists and describes the control buttons:

Table 2.3 Control Buttons

Buttons	Description
🔍+	Zoom in – Increases the detail level of the view. The whole picture might not fit the viewing area.
🔍−	Zoom out – Decreases the detail level of the view. Processor Expert tries to fit the whole picture to the viewing area.
↻	Rotate – Rotates the package clockwise.
▦	Resources (*available for BGA type packages only*) – Selects Resources view mode that shows a top side of the package without pins but including list of peripherals and showing their allocation by components.

Table 2.3 Control Buttons

Buttons	Description
▦	Pins Bottom (*available for BGA type packages only*) – Selects Pins view mode that shows a bottom side of the package with pins. The peripherals are not shown in this mode beacause the surface is covered with pins.
▦	Pins Top – Selects Pins view mode that shows a top side of the package with pins.

Pins

The following information about each pin is displayed on the processor picture:

(in case of BGA type package the pins are displayed only in the Pins view mode)

- pin name (default or user-defined)
- icon of a component that uses (allocates) the pin
- direction of the pin (input, output, or input/output) symbolized by blue arrows if a component is connected
- With new pin model (supported for few derivatives only), the background color of the pin reflects routing of the pin to the peripheral.

Pin names are shortened and written either from left to right or from top to bottom and are visible only if there is enough space in the diagram.

Some signals and peripherals cannot be used by the user because they are allocated by special devices such as power signals, external, or data bus. The special devices are indicated by a special blue icons. The allocation of peripherals by special devices can be influenced by processor component settings.

In case of BGA package, the pins that are used by some component are colored yellow. Move the cursor on the pin to get detailed information.

Hints

Pin hint contains:

- number of the pin (on package)
- both names (default and user-defined)
- owner of the pin (component that allocates it)
- short pin description from processor database

Component icon hint contains:

- component name
- component type
- component description

Shared Pins

If a pin is shared by multiple components, the line connecting the pin to the component has a red color. Refer to the Pin Sharing topic for details.

On-chip Peripherals

The following information about each on-chip peripheral is displayed on the processor package:

- peripheral device name (default or user-defined)
- icon of the component that uses (allocates) the peripheral device

Peripheral device hint contains:

- peripheral device name
- owner of the pin (component that allocates it)
- short peripheral device description

Hint on icon contains:

- component name
- component type
- component description

If a peripheral is shared by several components (for example, several components may use single pins of the same port), the icon is displayed.

NOTE Some peripherals work in several modes and these peripherals can be represented by a several devices in the processor databases. For example, the device "`TimerX_PPG` and "`TimerX_PWM` represents TimerX in the PPG and PWM mode. These devices can be displayed on the processor package, but they are also represented as a single block in the microcontroller block diagram.

Peripheral or Pin Pop-up Menu

The following commands are available in the pop-up menu:

- **Zoom in** — Increases the detail level of the view. The whole picture might not fit the viewing area.
- **Zoom out** — Decreases the size of the picture and detail level of the view.
- **Rotate** — Rotates the package by 90 degrees.
- **Pin Functional Properties** — This command is supported only if PinSettings component is supported for the selected processor. It adds PinSettings component, if it is not added in the project. It allows you to configure **Pin Functional Properties** for the pin. The selected values are written directly to PinSettings component for the pin.
- **Show Peripheral Initialization —** shows initialization values of all control, status and data registers. This option is supported for all devices displayed on a processor package. Refer to the Configuration Registers View for details.
- **Add Component/Template** — adds a component or template for the appropriate peripheral; all available components and templates suitable for the selected peripheral are listed. The components and templates in the list are divided by a horizontal line. It is possible to add only components or templates which are applicable for the peripheral. It means that is possible to add the component or template only if the peripheral is not already allocated to another component or components. The components/templates that cannot be added to the peripheral are grayed in the pop-up menu as unavailable. This option is supported for all devices displayed on processor package.
- **Remove Component** — allows to remove all components allocating peripheral in Processor view. Processor component cannot be removed.

Memory Map View

Figure below shows the processor address space and internal and external memory mapping. Detailed information for an individual memory area is provided as a hint when you move the cursor over it.

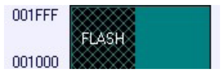
Table 2.4 Legend

Legend	Description
⬜	white: non-usable space
🟦	dark blue: I/O space
🟦	blue: RAM
🟦	light blue: ROM, OTP or Firmware
🟩	cyan: FLASH memory or EEPROM. This area can also contain a flash configuration registers area.
⬛	black: external memory

The address in the diagram is increasing upwards. To improve the readability of the information, the size of the individual memory blocks drawn in the window are different from the ratio of their real size (small blocks are larger and large blocks are smaller).

The black line-crossed area shows the memory allocated by a component or compiler. The address axis within one memory block goes from the left side to the right (it means that the left side means start of the block, the right side means the end).

Figure 2.28 Sample Of Used Part Of The Memory Area



Configuration Registers View

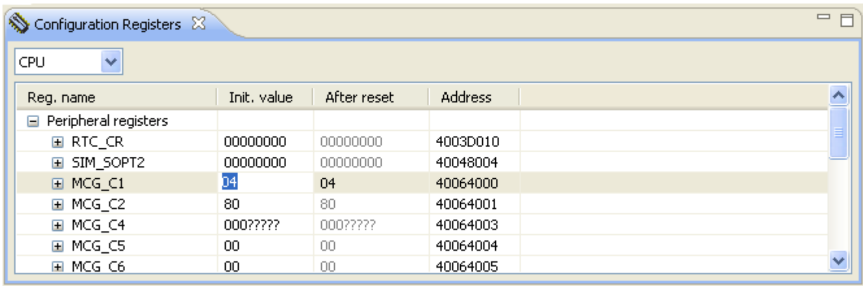
Configuration Registers view shows overview of the peripheral initialization settings for the current target microcontroller. It displays initialization values of all control, status, and data registers of selected peripheral/device including single bits. The values are grouped into two parts: **Peripheral registers** containing registers directly related to the selected peripheral/device and **Additional registers** containing the registers that are influenced by the component but are not listed for the peripheral currently selected in this view.

The initialization information reflects:

- Microcontroller default settings — When the peripheral is not utilized by any Embedded Component.

- Embedded Component settings — When the peripheral is utilized by an Embedded Component and the component settings are correct. Peripheral Initialization Inspector shows initialization as required by the component settings.

Figure 2.29 Configuration Registers View



Reg. name	Init. value	After reset	Address
Peripheral registers			
RTC_CR	00000000	00000000	4003D010
SIM_SOPT2	00000000	00000000	40048004
MCG_C1	04	04	40064000
MCG_C2	80	80	40064001
MCG_C4	000?????	000?????	40064003
MCG_C5	00	00	40064004
MCG_C6	00	00	40064005

The table shows the registers and their initialization value displayed in the column **Init. value**. You can modify the register value. Registers value can be changed if:

- They are not read-only and when the project is without errors
- Editing of configuration registers is supported by given component

This value written into the register or bit by the generated code during the initialization process of the application. It is the last value that is written by the initialization function to the register. The **After reset** column contains the value that is in the register by default after the reset.

The values of the registers are displayed in the hexadecimal and binary form. In case the value of the register (or bit) is not defined, an interrogation mark "?" is displayed instead of the value. The **Address** column displays the address of registers.

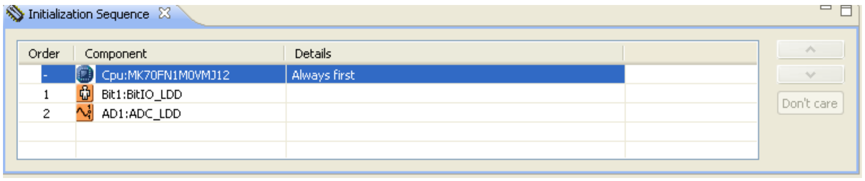
NOTE For some registers, the value read from the register after sometime can be different than the last written value. For example, some interrupt flags are cleared by writing 1. For details, refer to the microcontroller manual on registers.

In case the peripheral is allocated by a component and the setting of the component is incorrect, the initialization values are not displayed in the **Configuration Registers** view.

Initialization Sequence View

It is possible to customize initialization sequence of components. By default, the sequence is not specified. You can change the sequence using up/down buttons.

Figure 2.30 Initialization Sequence



Order	Component	Details
-	Cpu:MK70FN1M0VMJ12	Always first
1	Bit1:BitIO_LDD	
2	AD1:ADC_LDD	

Third column displays any conflicts or component specific messages. Initialization of processor component is always first. Disabled components are also listed (even if the code is not generated). This allows to create configuration for the disabled component to re-enable them. You can unspecify the initialization order of given component by clicking on the **Don’t care** button.

3

Application Design

This chapter will help you to design application using Processor Expert and Embedded Components. You will find here recommendations and solutions to write and optimize a code effectively.

This chapter explains:

- Creating Application using Processor Expert
- Basic Principles
- Configuring Components
- Implementation Details
- Code Generation and Usage
- Embedded Component Optimizations
- Converting Project to Use Processor Expert
- Low-level Access to Peripherals
- Processor Expert Files and Directories
- Static Code Support in Processor Expert
- Internal signals

Creating Application using Processor Expert

You can create new project using project wizard for Processor Expert application that support many targets and also for MQXLite project that support only Kinetics target.

To create an application using processor expert:

1. **Open an example**

   You can start learning Processor Expert by opening one of the available examples. Select **File > Import** to open the **Import** dialog. Then select **General > Existing Projects** into workspace and click **Next**.

2. In the **Import Projects** screen, click the **Browse** button and select the directory of the sample project that you want to use under this folder.

3. Click **Finish**.

4. **Code generation**

   After opening an example, invoke the code generation of the project to obtain all sources. In the project tree, right-click the `ProcessorExpert.pe` file and select the **Generate Processor Expert Code** command. The generated code is placed in the **Generated_Code** sub-folder of the project.

   The Processor Expert views can be opened any time using the menu command **Processor Expert > Show Views**.

NOTE Refer to the Processor Expert Tutorials topic for step-by-step tutorials on creating Processor Expert projects from the beginning.

Basic Principles

The application created in Processor Expert is built from the building blocks called Embedded Components. The following topics describe the features of the Embedded Components and the processor components that are special type of Embedded Components and what they offer to the user.

- Embedded Components
- Processor Components

Embedded Components

Embedded components encapsulate the initialization and functionality of embedded systems basic elements, such as microcontroller core, on-chip peripherals, (for details on categories of components delivered with Processor Expert, refer to the Component Categories topic) FPGAs, standalone peripherals, virtual devices, and pure software algorithms.

These facilities are interfaced to the user through properties, methods and events. It is very similar to objects in the Object Oriented Programming (OOP) concept.

Easy Initialization

You can initialize components by setting their initialization properties in the Component Inspector. Processor Expert generates the initialization code for the peripherals according to the properties of the appropriate components. You can decide whether the component will be initialized automatically at startup or manually by calling the component's Init method.

Easy On-chip Peripherals Management

Processor Expert knows exactly the relation between the allocated peripherals and the selected components.

When you select a peripheral in the component properties, Processor Expert proposes all the possible candidates but signals which peripherals are allocated already (with the icon of the component allocating the peripheral). PE also signalizes peripherals that are not compatible with the current component settings (with a red exclamation mark). In case of an unrealizable allocation, an error is generated.

Unlike common libraries, Embedded Components are implemented for all possible peripherals with optimal code. The most important advantages of the generated modules for driving peripherals are that you can:

- Select any peripheral that supports component function and change it whenever you want during design time.
- Be sure that the component setting conforms to peripheral parameters.
- Choose the initialization state of the component.
- Choose which methods you want to use in your code and which event you want to handle.
- Use several components of the same type with optimal code for each component.

The concept of the peripheral allocation generally does not enable sharing of peripherals because it would make the application design too complicated. The only way to share resources is through the components and their methods and events. For example, it is possible to use the RTIshared component for sharing periodic interrupt from timers.

Methods

Methods are interfacing component functionality to user's code. All enabled methods are generated into appropriate component modules during the code generation process. All Methods of each component inserted into the project are visible as a subtree of the components in the **Components** view.

You can use in your code all enabled methods. The easiest way to call any method from your code is to drag and drop the method from **Components** view to the editor. The complexity and number of methods depend on the component's level of abstraction.

Events

Some components allow handling the hardware or software events related to the component. You can specify the name on function invoked in the case of event occurrence. They are usually invoked from the internal interrupt service routines generated by Processor Expert. If the enabled event handling routine is not already present in the event module then the header and implementation files are updated and an empty function (without any code) is inserted. You can write event handling code into this procedure and this code will not be changed during the next code generation.

All Methods and Events of each component inserted into the project are visible as a subtree of components in the **Components** view.

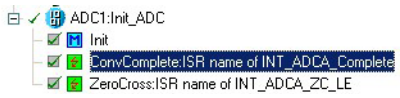
Interrupt Subroutines

Some components, especially the low-level components and the Peripheral Initialization components (refer to more details in Component Categories topic) allow to assign an interrupt service routine (ISR) name to a specific interrupt vector setup.

The name of the Interrupt service is generated directly to the interrupt vector table and you have to do all necessary control registers handling within the user code. Refer to the Typical Usage of Component in User Code topic for details.

ISRs items are listed in the subtree of a component in the **Components** view. Empty Interrupt Service Routines (ISR) can be removed from the event module.

Figure 3.1 Example Of a Component With Two ISRs



ADC1:Init_ADC
Init
ConvComplete:ISR name of INT_ADCA_Complete
ZeroCross:ISR name of INT_ADCA_ZC_LE

Highly Configurable and Extensible Library

Embedded Components can be created and edited manually or with the help of CDE. Processor components are a special category of components.

Component Categories

Complete list of the component categories and corresponding components can be found in the Component Categories page of the Components Library View.

The components are categorized based on their functionality, so you can find an appropriate component for a desired function in the appropriate category.

These are the following main categories, which further contain various sub-categories.

- **Processor External Devices** — Components for devices externally controlled to the processor. For example, sensors, memories, displays or EVM equipment.
- **Processor Internal Peripherals** — Components using any of on-chip peripherals offered by the processor. The Components Library folder with the same name contains sub-folders for the specific groups of functionality. For example, Converters, Timers, PortIO.

NOTE It seems that components (especially in this category) correspond to on-chip peripherals. Even this declaration is close to true, the main purpose of the component is providing the same interface and functionality for all supported microcontrollers. This portability is the reason why the component interface often doesn't copy all features of the specific peripheral.

- **Logical Device Drivers** — LDD components. Refer to the Logical Device Drivers topic for details.
- **Operating systems** — Components related to Processor Expert interaction with operating system running on the target.
- **SW** — Components encapsulating a pure software algorithms or inheriting a hardware-dependent components for accessing peripherals. These components (along with components created by the user) can be found in a components library in the folder `SW`.

Specific functionality of the microcontroller may be supported as a **version-specific** settings of the component. For more information about this feature, refer to the Version specific parts in the component documentation or Components Implementation Details topic.

Levels of Abstraction

Processor Expert provides components with several levels of abstraction and configuration comfort.

- **LDD Components** — Logical Device Drivers. The LDD components are efficient set of components that are ready to be used together with RTOS. They provide a unified hardware access across microcontrollers allowing to develop simpler and more portable RTOS drivers or bare board application. Refer to the Logical Device Drivers topic for details.
- **High Level Components** — Components that are the basic set of components designed carefully to provide functionality to most microcontrollers in market. An application built from these components can be easily ported to another microcontroller supported by the Processor Expert. This basic set contains for example components for simple I/O operations (BitIO, BitsIO, ByteIO, ...), timers (EventCounter, TimerInt, FreeCntr, TimerOut, PWM, PPG, Capture, WatchDog,...), communication (AsynchroSerial, SynchroMaster, SynchroSlave, AsynchroMaster, AsynchroSlave, IIC), ADC, internal memories.

  This group of components allows comfortable settings of a desired functionality such as time in `ms` or frequency in `Hz` without user knowing about the details of the hardware registers. microcontroller specific features are supported only as processor specific settings or methods and are not portable.

The components inheriting or sharing a high-level component(s) to access hardware are also high-level components.

- **Low Level Components** — Components that are dependent on the peripheral structure to allow you to benefit from the non-standard features of a peripheral. The level of portability is decreased due to a different component interface and the component is usually implemented only for a microcontroller family offering the appropriate peripheral. However, you can easily set device features and use effective set of methods and events.
- **Peripheral Initialization Components** — Components that are on the lowest level of abstraction. An interface of such components is based on the set of peripheral control registers. These components cover all features of the peripherals and are designed for initialization of these peripherals. Usually contain only `Init` method, refer to the Typical Usage of Peripheral Initialization Components topic for details. The rest of the function has to be implemented using a low level access to the peripheral. This kind of components are located at: **processor Internal Peripherals/ Peripheral Initialization Components** of the components library and they are available only for some processor families. The interface of these components might be different for a different processor. The name of these components starts with the prefix '`Init_`'.

Table 3.1 Features of Components at Different Level of Abstraction

Feature	LDD Components	High level	Low level	Peripheral Init
High-level settings portable between different microcontroller families	partially	yes	partially	no
Portable method interface for all processor families	yes	yes	partially (usually direct access to control registers)	*Init* method only
Processor specific peripheral features support	mostly yes	partially	mostly yes	full
Low-level peripheral initialization settings	partially	no	partially	yes
Speed mode independent timing	yes	yes	mostly yes	no

Table 3.1 Features of Components at Different Level of Abstraction

Feature	LDD Components	High level	Low level	Peripheral Init
Events support	yes	yes	yes	no (direct interrupt handling)
Software emulation of a component function (if the specific hardware is not present)	no	yes	no	no
Support for RTOS drivers creation	yes	no	no	no

Logical Device Drivers

Logical Device Drivers were developed to offer users the Hardware Abstraction Layer (HAL) for bare-metal applications as well as RTOS applications. The components provide tested, optimized C code tuned to the application needs. The code may be tuned to the specific RTOS when the RTOS component is in the project.

Differences Between LDD and High Level Components

- Each component provides `Init()` method to initialize appropriate peripheral and driver. `Init()` method returns a pointer to driver's device structure.
- Each component provides `Deinit()` method to de-initialize appropriate peripheral and driver.
- The first parameter of each component's method is a pointer to a device structure returned from `Init()` method. It is up to you to pass a valid device structure pointer to component's methods (null check is highly recommended).
- The `Init()` method has one parameter `UserDataPtr`. You can pass a pointer to its own data and this pointer is then returned back as a parameter in component's events. The pointer or date pointed by this pointer is not modified by driver itself. A bare-board application typically passes a null pointer to `Init()` method.
- LDD components are not automatically initialized in processor component by default. If `Auto initialization` property is not enabled, you must call appropriate `Init()` method during runtime. Otherwise the Init method is automatically called in processor component and device structure is automatically defined.

- LDD components have RTOS adapter support allowing to generate variable code for different RTOSes.
- Runtime enable/disable of component events.
- Low Power Modes support.

Logical Device Drivers in Bare-metal Applications

Logical Device Drivers can be used in applications where the RTOS is not required. Logical Device Drivers in bare-metal environment have following specific features:

- `Init()` method of each component uses a static memory allocation of its device structure.
- Interrupt Service Routines are statically allocated in generated interrupt vector table (IVT).
- The Linker Command File (LCF) is generated from processor component.
- The main module (ProcessorExpert.c) is generated.

Logical Device Drivers in RTOS Environment

Logical Device Drivers in RTOS environment have following specific features:

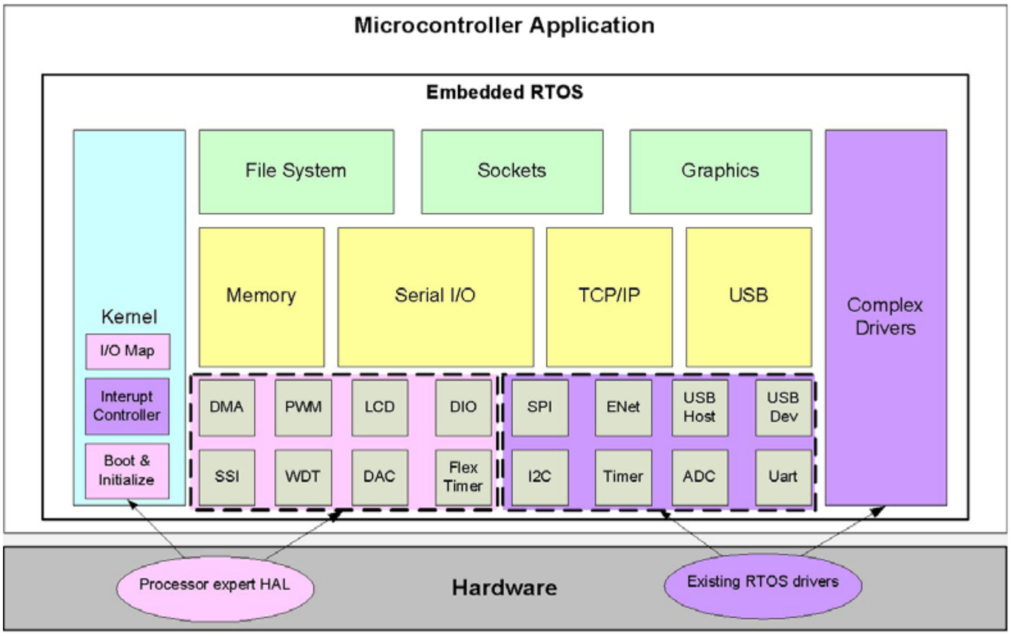
- `Init()` method of each component uses a dynamic allocation of its device structure through the RTOS API.
- `Deinit()` method of each component uses a dynamic de-allocation of its device structure through the RTOS API.
- Interrupt Service Routines are allocated through the RTOS API in `Init()` method and de-allocated in `Deinit()` method of each component.
- The Interrupt table is not generated from processor component in case whether RTOS provides runtime allocation of interrupt service routines.
- The Linker Command File (LCF) is not generated from processor component in case that RTOS provides its own LCF for applications.
- The main module (ProcessorExpert.c) is not generated if specified in RTOS adapter.

For information and hints on LDD components usage, refer to the Typical LDD Components Usage topic for details.

RTOS Adapter

The RTOS adapter component is a way how to utilize generated code to the specific RTOS. The RTOS adapter provides necessary information to the driver which API should be used to allocate memory, create a critical section, allocate interrupt vector.

Figure 3.2 Example of HAL Integration into Existing RTOS



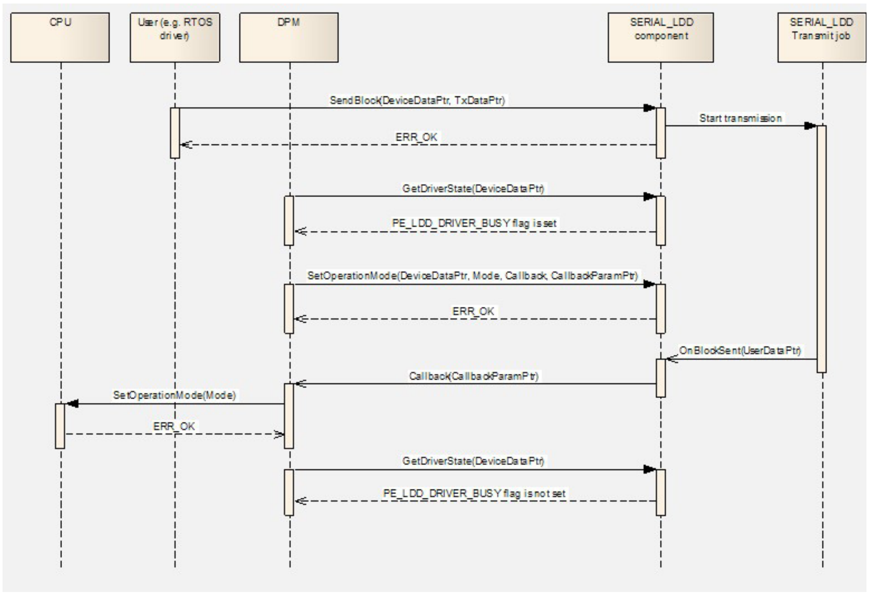
Shared Drivers

Logical device drivers support the shared mode that means that more components can be put together to provide one instance of API. You can access each component instance through the API of the shared component. A driver device data structure is used for resolution which peripheral instance shall be accessed. Currently there are three components that support shared mode: `Serial_LDD`, `CAN_LDD` and `Ethernet_LDD`.

Low Power Features

Logical device drivers in conjunction with processor component implement low power features of a target microcontroller. Each LDD component define two methods related to the low power capability – `SetOperationMode()` and `GetDriverState()`. For more details, refer to the documentation of components.

Figure 3.3 Usage of Low Power API in Logical Device Drivers


CPU
User (e.g. RTOS driver)
DPM
SERIAL_LDD component
SERIAL_LDD Transmit job
SendBlock(DeviceDataPtr, TxDataPtr)
Start transmission
ERR_OK
GetDriverState(DeviceDataPtr)
PE_LDD_DRIVER_BUSY flag is set
SetOperationMode(DeviceDataPtr, Mode, Callback, CallbackParamPtr)
ERR_OK
OnBlockSent(UserDataPtr)
Callback(CallbackParamPtr)
SetOperationMode(Mode)
ERR_OK
GetDriverState(DeviceDataPtr)
PE_LDD_DRIVER_BUSY flag is not set

In the example above, DPM (Dynamic Power Manager) task may opt to care for a selected number of peripherals for graceful power mode change (for example, FEC, CAN) and rest of the peripheral drivers need not know the power mode change. When opted for informing a peripheral device driver, the DPM can build a semaphore object for low power acknowledgement from the device drivers. When all such acknowledgements arrive (ie. Semaphore count equals zero) the processor can be placed into a wait/sleep power mode. In the future, with silicon design modifications, these semaphores can be implemented in the hardware and as a result a much faster power mode change can be expected. There is no DPM in typical bare-metal applications the DPM task is implemented. In this case, DPM is substituted by a user application code.

Processor Components

A processor component is an Embedded Component encapsulating one processor type. A Processor Expert project may contain one or more processor components. The project generated for one processor is called an application. Processors included in a project are displayed in the upper part of the **Components** view. It is possible to switch among the processor component, but only one of the processor can be active at one time.

The **Build options** accessible in the Component Inspector of the processor component allow you to set properties of the **Compiler** and **Debugger** (if it is supported).

Portability

- It is possible to change the target microcontroller during the development of an application and even to switch between multiple microcontrollers. This can be done simply by adding another processor to the project and selecting it as the target processor.
- To connect the new processor peripherals to the application components correctly, it is possible to specify the processor on-chip peripheral names. This way the same peripheral could be used on different processor derivatives even if the original name is different.

Adding a Processor to a Project

1. In the **Components Library** view, select the **Processors** tab and find the desired processor component.
2. Double-click the desired processor icon to add it to the project. When the processor component is added, it appears in the upper part of the **Components** view. If selected as the target processor, the processor will be displayed in the **Processor** view.

Selecting a Processor as Target Processor

The first microcontroller added to the project is automatically selected as the target processor. It means that code will be generated for this microcontroller. When there are more than one processor in the project, the target processor can be changed by following these steps:

1. Right-click the processor icon in the **Components** view to display a pop-up menu.
2. Select the **Select processor as target** option, the processor is selected as target.

This setting doesn't affect the setting of the target. If user changes the target processor in the Components view and the processor doesn't match with the current target settings, the **Linker** dialog box is invoked during the code generation allowing user to update the linker setup.

Changing Settings

To modify the processor component settings (its properties, methods, events, external bus, timing, user-reserved peripherals, compiler and debugger settings) is to invoke the Inspector for the selected processor component.

If you have added processor to your project, you can invoke Component Inspector by performing either of the following:

- Right-click the processor icon in the **Components** view to display pop-up menu and select the **Component Inspector** view.

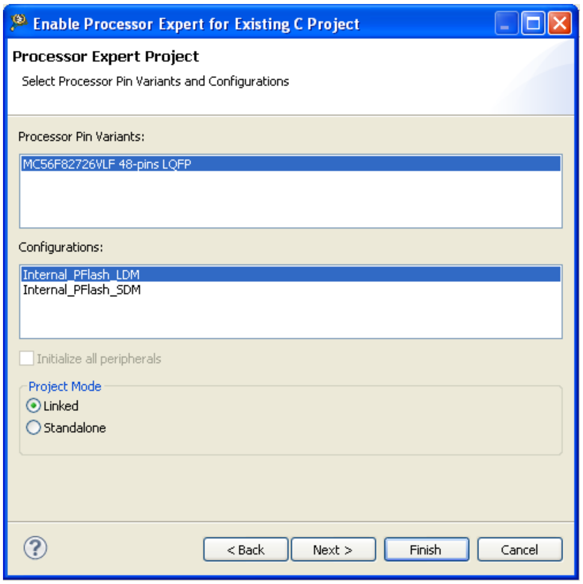
- Double-click the processor icon in the **Components** view.

For a detailed description of the current processor properties, methods and events, select **Help on Component** command in the **View** menu (drop-down arrow) in the **Component Inspector** view.

Processor Component Variants Selection

This dialog is shown during the creation of a new project using Project Wizard or when you add a new processor component into project using **Components Library** view.

Figure 3.4 Processor Component Variants Selection


Enable Processor Expert for Existing C Project

Processor Expert Project

Select Processor Pin Variants and Configurations

Processor Pin Variants:

MC56F82726VLF 48-pins LQFP

Configurations:

Internal_PFlash_LDM

Internal_PFlash_SDM

[ ] Initialize all peripherals

Project Mode

(●) Linked

( ) Standalone

In this dialog, you can select processor pin-variants and configurations that will be supported for switching later in the project. Each selection of variant or configuration represent one processor component pre-set. For example, if you select two pin variants and two configuration, there will be four processor components added into the project.

If you have selected **Initialize all peripherals** checkbox, it adds all initialization components to the project for all supported peripherals.

NOTE This option is not supported for all derivatives. If supported on given family, the project can contain except the CPU component and the PinSettings component for configuring pin routing and electrical properties.

The project wizard offers support for CPUs that how the static files are used in the Processor Expert project. There are two project modes:

- In Linked mode, static files (for instance, cpu and peripheral init modules, PDD modules, io map, system files) are linked from the repository of Processor Expert (`ProcessorExpert\lib\subdirectory`). Modification of these files is possible only in the Processor Expert's repository and affects other projects.
- In Standalone mode, static files (for instance, cpu and peripheral init modules, PDD modules, io map, system files) are placed in the project folder. They are copied from Processor Expert's repository (`ProcessorExpert\lib\subdirectory`) during project creation. This mode allows to modify the static files in the project without affecting other projects.

NOTE Static files are not supported for all derivatives.

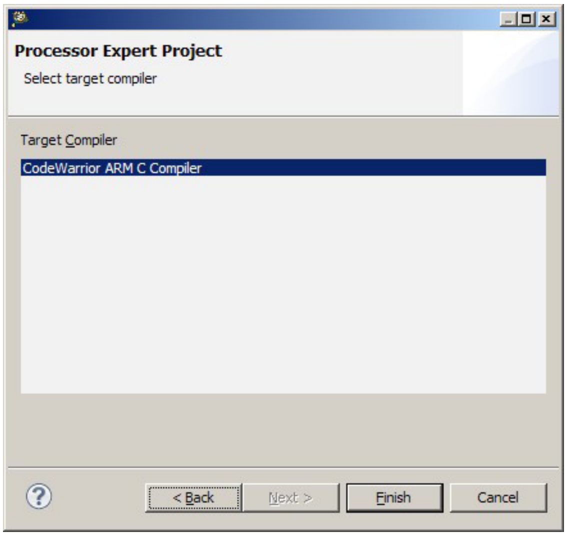
For details on configurations, refer to the Configurations topic.

Compiler Selection

This dialog is shown when you add a new processor component into project using **Components Library** view.

If there are more target compilers available, you can select the compiler to be used for the newly added processor.

Figure 3.5 Compiler Selection


Processor Expert Project
Select target compiler

Target Compiler
CodeWarrior ARM C Compiler

Processor Properties Overview

Processor Properties can be set in processor **Component Inspector** view. The complete list of processor properties and their description is available in the help page for the processor. To open the processor help page, select **Help > Help on Component** from the menu bar in the **Component Inspector** view.

Following properties define the basic settings of the processor:

- Processor type
- External Xtal frequency (and sub-clock xtal frequency)
- PLL settings
- Initialization interrupt priority
- External bus and signals
- Speed modes (refer to the Speed Modes Support topic).
- All other functions that are not directly encapsulated by components

Speed Modes Support

NOTE Speed Modes are not available for Kinetis and ColdFire+ family microcontrollers.

The processor component supports up to three different speed modes. The three speed modes are Processor Expert specific concept which (among all the other PE features and concepts) ensures the portability of the PE projects between different processor models.

In fact, the three speed modes are a generalization of all the possible processor clock speed modes used for power-saving that can be found in most of the modern microcontrollers. In the area of embedded systems, power saving and power management functions are so important that you can not neglect the proper HW- independent software implementation of these functions.

Therefore, for keeping the portability (HW independence) of PE projects, it is recommended not to program the processor speed functions manually, but use these three processor Component speed modes instead:

- **High speed mode** — this mode is selected after reset and must be enabled in the project. This speed mode must be the fastest mode of the main processor clock.
- **Low speed mode** — this mode is usually used for another PLL or main prescaler settings of the main processor clock.
- **Slow speed mode** — this mode is usually used for the slowest possible mode of the main processor clock.

Switching Speed Modes at Runtime

The modes can be switched in the runtime by the following processor component methods:

- SetHighSpeed
- SetLowSpeed
- SetSlowSpeed

If a speed mode is enabled in the processor Component properties, the corresponding method is enabled automatically.

NOTE It is highly recommended to disable interrupts before switching to another speed mode and enable them afterwards.

Speed Modes Support in Components

Using the component property processor clock/speed selection, it is possible to define the speed modes supported by the component.

Some components allow to set two values of the processor clock/speed selection property:

- **Supported** — The processor clock/speed selection group contains properties defining which speed modes are supported for the component.
- **Ignored** — The speed mode settings are ignored and there are no action is performed when a speed mode is changed, that is the peripheral continues to run with the same configuration for all speed modes. No speed mode handling code is generated for this component. The component timing values are valid only for high-speed mode.

The following features are available for high-level components, if the processor clock/speed selection is not set to ignored:

- During the design, all the timing-related settings for such a component are checked to be correct in all the speed modes that the component supports and the component is enabled in these modes.
- If the speed mode is changed, the current timing components are preserved (recalculated to be the same in the new speed mode), except the timing that is set at runtime from interval. If the processor speed mode is changed to a mode that the component does not support for any reason, the component is disabled right after the processor speed mode is changed. Otherwise, the component is enabled.
- Before or after the speed mode is changed, the `BeforeNewSpeed` and `AfterNewSpeed` event functions are called.

Clock Configurations Support

Clock configurations are available for Kinetis, Vybrid, Cold Fire+ and MC56F82xxx families. As in case of speed modes, the purpose of clock configuration is to offer power management functionality while ensuring project portability. This topic explains the concept and its application in LDD components.

- Clock Configuration
- Multiple Clock Configuration
- Clock Configuration 0
- Support of Clock Configurations in Embedded Components

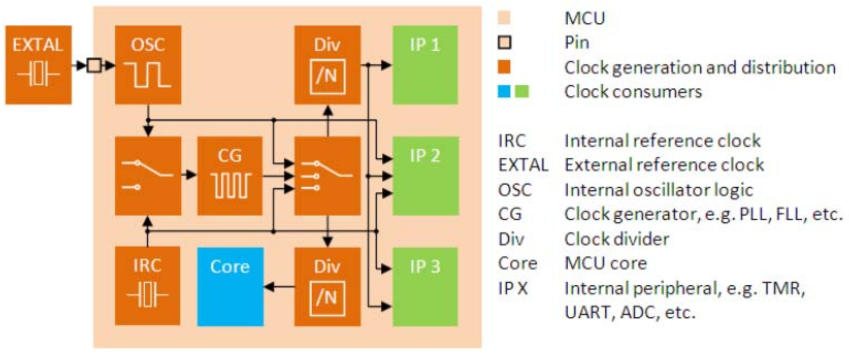
Clock Configuration

Typical processor or microcontroller allows multiple reference clock sources usage, contains clock synthesis modules to process these reference clocks and distribute them across the system. This system timing scheme provides a set of clocks produced by the system modules and is described as a clock path which starts at selected reference clock source, such as internal oscillator or external crystal, and ends at variety of processor internal clocks, such as core (instruction) clock or other clocks consumed by internal peripherals.

Clock configuration is feature allowing design time creation and management of multiple system timing schemes and is handled by a CPU (processor) component. This feature is derived from the Speed modes used by the high-level components. For more details, see Speed Modes Support topic.

The following schematic illustrates a simple system timing scheme. It consists of internal and external reference clock, general clock generator module and several internal modules that can consume variety of clocks. The orange block represents modules involved in clock generation and distribution whereas, blue and green blocks denote clock consumers. For example, core module and all internal peripherals can be clocked from divided clock generator output or from divided internal or external reference clock. Additionally, internal peripheral 3 can be fed directly from internal reference clock and internal peripheral 2 also directly from external reference.

Figure 3.6 System timing scheme example


EXTAL
OSC
Div
/N
IP 1
CG
IP 2
IRC
Core
Div
/N
IP 3
MCU
Pin
Clock generation and distribution
Clock consumers
IRC Internal reference clock
EXTAL External reference clock
OSC Internal oscillator logic
CG Clock generator, e.g. PLL, FLL, etc.
Div Clock divider
Core MCU core
IP X Internal peripheral, e.g. TMR, UART, ADC, etc.

In Processor Expert, the orange blocks represents clock configuration and the green blocks represents embedded components. As mentioned, clock configurations are configured in the CPU (processor) component.

There user can make selection of used reference clock and specify configuration of clock generator modules and dividers. Finally, in CPU component user always select frequency of all major processor clocks, such as core, system or flash memory frequencies that is clock, which define speed of the whole system, or peripheral bus clocks, which influence timing characteristics of chip internal peripherals. embedded components relies on these values and use them as the input value for timing features they encapsulate. The relationship between clock configurations set in the CPU (processor) component and embedded components is explained in Support of Clock Configurations in Embedded Components topic.

There is always at least one clock configuration in the project - Clock configuration 0. However, it is possible to use more clock configurations in the same project. For example, use case of multiple clock configuration is in Multiple Clock Configuration topic.

Multiple Clock Configuration

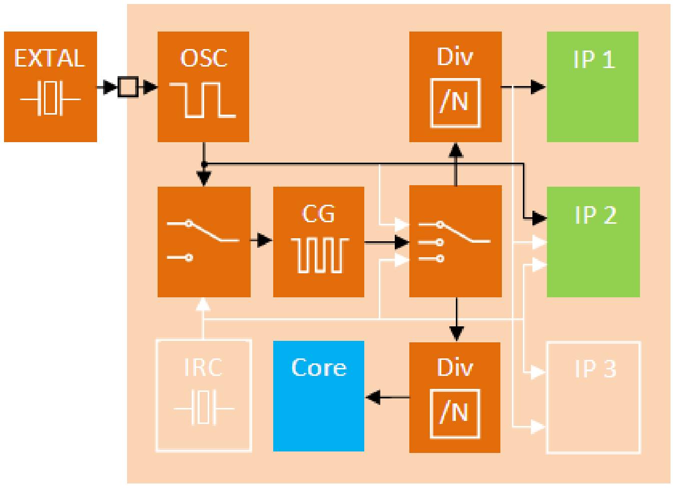
Every Processor Expert project uses at least one clock configuration to set main system clock's configuration and distribution. However, it is possible to predefine up to eight (in case of Speed modes it was up to three ones) different timing schemes and switch between them during runtime by `SetClockConfiguration()` CPU (processor) method call.

This is useful in case of low-power oriented applications as there can be separate highspeed (when fast computation is needed) and low-power timing schemes (when chip goes idle) and application can easily switch between them.

The following example is a use case of simple battery-powered application which will use two clock configurations. In this scenario, the application will put the microcontroller in low-power state, waiting for external or internal event (such as button push or internal timer overflow). When the event occurs, microcontroller is switched to high-speed mode to make a measurement, computation and communication with external devices. When the job is done, microcontroller is switched back to sleep.

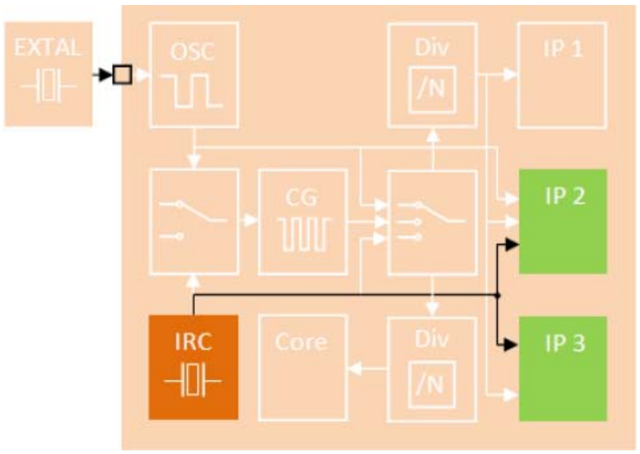
The figure below shows situation when application runs, the microcontroller is clocked from an external source, the reference clock is processed by internal clock circuits (forexample, clock frequency is increased by PLL module) and distributed across system. Core and IP 1 run from divided output of internal clock generator (data computation and communication interface handling) whether IP 2 runs directly from external reference clock (for example, because frequency from clock generator output is out of admissible range of this peripheral). Internal reference clock and IP 3 are not used therefore are powered down.

Figure 3.7 Clock configuration example in application run state



The next schematic shows low-power configuration, an internal clock reference is used instead of external as it spends less energy. No computation power is needed nor any peripheral that would need high speed clocks enabled, therefore core is powered off along with the clock generator circuits. Only IP 2 and IP 3 are enabled and clocked directly from the internal clock source, waiting for event that will bring the microcontroller back to operational mode.

Figure 3.8 Clock configuration example in application low-power state



This example also shows that some peripherals allow using multiple clock sources (for instance clock generator output, raw output from reference clock source or other alternative clocks). Clock configuration in such case can be used to switch clock source of the peripheral clock input. Furthermore, the example demonstrates possibility to enable or shut down entire peripheral during clock configuration switch. For more information about clock configuration options available in the embedded components, see Support of Clock Configurations in Embedded Components topic. Clock configuration switch is explained in the topic below.

Changing active clock configuration

CPU (processor) component `SetClockConfiguration(ConfigurationID)` method is used to change active clock configuration. This method will switch to clock configuration defined by `ConfigurationID` passed as input parameter. The required configuration has to be predefined in the processor component, otherwise the method returns an error.

Clock configuration change is done during CPU `SetClockConfiguration()` method call in the following order:

1. It is checked whether required clock configuration number was defined in the CPU (processor) component. If not, method returns with an error.
2. System timing is configured according to new clock configuration, including reference clock source selection, clock generator modules and system dividers.
3. All LDD components which support clock configuration feature are called to configure their underlying peripherals - global internal `LDD_SetClockConfiguation()` method is invoked which in return invokes internal `SetClockConfiguration()` method of the LDD components.

Note that during the transition period glitches at the peripheral outputs may occur. Therefore, it is recommended to temporarily disable the peripherals before the clock configuration is switched to prevent any unpredictable behavior.

Clock Configuration 0

Unlike other clock configurations, Clock Configuration 0 has some specific features:

- Clock Configuration 0 is always in the project. Each project has to have at least one clock configuration.
- Application execution always starts in the Clock Configuration 0. Before `main()` function execution starts, startup code of Processor Expert initializes the system timing according to settings of this clock configuration.
- All Peripheral Initialization Components timing settings relates to Clock Configuration 0 (see Levels of Abstraction topic). Because multiple clock configurations are not supported in initialization components, timing-related values or information showed in the initialization components are valid for initial system timing. For example, if a specific timer period value is calculated from current divider property settings and showed as additional information in the component then this time value is valid only for Clock Configuration 0 and might be different when the clock configuration is switched.

Support of Clock Configurations in Embedded Components

As described in Clock Configuration topic, particular system timing scheme, which is defined as reference clock source selection and settings, clock processing settings (for example, configuration of a PLL circuit) and value of main system clocks (such as internal peripherals bus clock), is in Processor Expert represented as a clock configuration and is configured in CPU (processor) component. In case of embedded components that encapsulate timing features derived from the system timing, Processor Expert also ensures that component timing settings are valid and consistent with the clock configuration. Unlike other clock configurations, Clock Configuration 0 has some specific features:

NOTE As mentioned in Clock Configuration 0 topic, the Peripheral Initialization Components (see Levels of abstraction topic) support only Clock Configuration 0. Therefore, following part of documentation generally applies to Logical Device Driver (LDD) components and High Level Components (although implementation of described features slightly differs in case of high-level components as they operate with legacy speed modes conventions).

The following table shows number of supported clock configurations for each category of embedded component.

Table 3.2 Supported Clock Configurations

Component Category	Clock Configurations	Note
Peripheral Initialization	1	Clock configuration 0
High-level	3	Speed modes
LDD	8	

There are two main clock configuration features available in embedded components:

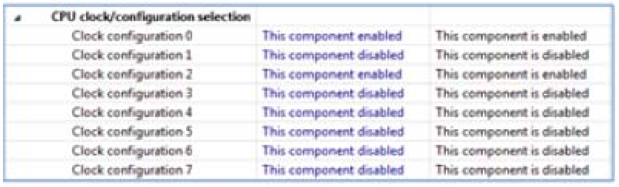
- Enabled clock configurations selection
- Timing settings

Enabled clock configurations selection

For each clock configuration, it is possible to specify whether component will be enabled or disabled. This means that when application switches to the particular clock configuration, using the `SetClockConfiguration()` CPU method call, the underlying internal peripheral will be switched on or off (if the hardware allows it). This way, clock configurations can be utilized to create not only multiple system timing schemes, but also power consumption-oriented scenarios.

The figure of **Component Inspector** view displays the clock configuration selection group available in LDD components.

Figure 3.9 Clock configuration selection in LDD component



CPU clock/configuration selection		
Clock configuration 0	This component enabled	This component is enabled
Clock configuration 1	This component disabled	This component is disabled
Clock configuration 2	This component enabled	This component is enabled
Clock configuration 3	This component disabled	This component is disabled
Clock configuration 4	This component disabled	This component is disabled
Clock configuration 5	This component disabled	This component is disabled
Clock configuration 6	This component disabled	This component is disabled
Clock configuration 7	This component disabled	This component is disabled

This feature is not supported by components that don't control the whole peripheral. Typical example are pin manipulation components, such as `BitIO_LDD` or `ExtInt_LDD` which are related just to single pin from a port peripheral and multiple instances of these components can be used on the same port. Components that support this feature must have at least one enabled clock configuration.

Timing settings

Timing settings are vital part of functionality of some peripherals. For example, in case of timer peripheral, it would be period of counter overflow, conversion time of analog-todigital converter or baud rate of communication interface. Configuration of these features in the high-level and LDD components is done in the **Timing Settings** dialog. This user-friendly interface allows configuration of peripheral timing settings. Generally, user can specify required timing value directly or pick up the suitable value from list of values computed according to current settings. For detailed documentation, see Dialog Box for Timing Settings topic and Timing Settings topic.

Fixed timing value

Timing value is same for all clock configurations. The default behavior of embedded components is to keep the value set in the timing settings same for all clock configurations. Component will ensure that peripheral will be configured to achieve required timing value regardless of the timing scheme used in particular clock configuration. For example, timer component set to produce interrupt with specific period will configure underlying peripheral in such a manner that for each clock configuration in which the component is enabled it will produce the interrupt with the same period. Similarly, communication component will transfer data with the same baud rate in all enabled clock configurations.

Processor Expert contains timing model of each peripheral and check values entered by the user. If required value is not achievable for some particular clock configuration, user is already informed about that during the design time. This approach is common for all embedded components. However, some of the components additionally allow creating list of timing values and switching between them using dedicated, component-specific method. For more information about available types of timing values, see Runtime Setting Configuration topic.

Input clock source selection

Often peripherals can be clocked not only from internal peripheral bus, but also from variety of alternative clock sources, such as dedicated asynchronous clock generator or any other system clock source. Although, the way how such selection is implemented is component-specific and might be common for all clock configurations, it is common practice that high-level and LDD components allow selecting input clock source for each clock configuration independently, switching between them during the application runtime (during `SetClockConfiguration()` method call). If peripheral input clock is not available, for example, because the clock configuration is set to disable the required reference clock source, the Processor Expert notifies the user about this.

Configuring Components

Configuring the components in the project is one of the main activities in Processor Expert. It affects the initialization, run-time behavior and range of functionality available to the generated code. For the description of the user interface of the components settings, refer to the Components View and Component Inspector.

The following topics provide hints and information about how to configure the Embedded Components used in the project correctly and effectively.

- Interrupts and Events
- Configurations
- Design Time Checking: Consequences and Benefits
- Timing Settings
- Creating User Component Templates
- Signal Names
- Component Inheritance and Component Sharing
- Pin Sharing
- Export and Import

Interrupts and Events

It describes the details of interrupt and events processing in the code generated by Processor Expert.

An interrupt is a signal that causes the processor stop the execution of the code and execute the Interrupt service routine. When the execution of the code is suspended, the current state of the processor core is saved on the stack. After the execution finishes, the previous state of the processor core is restored from the stack and the suspended program continues from the point where it was interrupted. The signals causing interrupts can be hardware events or software commands. Each interrupt can have an assigned Interrupt Service Routine (ISR) that is called when the interrupt occurs. The table assigning the subroutines to interrupts is called Interrupt Vector Table and it is completely generated by Processor Expert. Most of the interrupts have corresponding Processor Expert Events that allow handling of these interrupts. Processor Expert allows to configure interrupt priorities, if they are supported by the processor. Refer to the Processor Expert Priority System for details.

Processor Expert Events are part of the Embedded component interface and encapsulate the hardware or software events within the system. Events are offered by the High and Low Level components to help you to service the events without any knowledge of the platform specific code required for such service.

Processor Expert Events can be enabled and disabled and have a user-written program subroutines that are invoked when the event occurs. Events often correspond to interrupts and for that case are invoked from the generated ISR. Moreover, the event can also be a software event caused by a buffer overflow or improper method parameter.

Interrupts Usage in Component's Generated Code

Some high-level components use interrupt service routines to provide their functionality. Usage of interrupts can usually be enabled/disabled through property Interrupt service/event. If the interrupt service is used, complete interrupt service routine is generated into component's driver and the generated code contains configuration of the corresponding peripheral to invoke the interrupts.

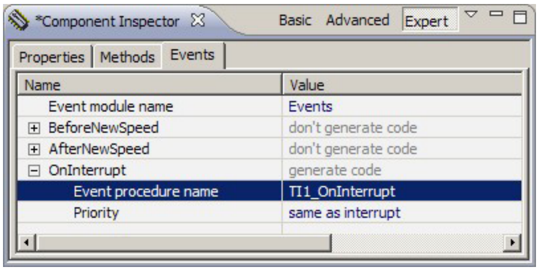
You should be careful while disabling an interrupt. If a component should operate properly, it is necessary to allow the interrupt invocation by having interrupts enabled and (if the processor contains priority system) set the corresponding interrupt priority level. This can be done using the appropriate method of the processor component.

NOTE It is a common bug in user code, if the application is waiting for a result of the component action while the interrupts are disabled. In this situation, the result of the component method does not change until the interrupt service routine is handled. Refer to the description of the property Interrupt service/event for detailed information about the particular component.

Enabling Event

Functionality of each event can be enabled or disabled. You can easily enable the event and define its name within the Component Inspector of the appropriate component. Another possibility is to double-click an event icon in the component's subtree or use a pop-up menu in the **Component Inspector** view.

Figure 3.10 Event Example in the Component Inspector Events Tab



Name	Value
Event module name	Events
⊞ BeforeNewSpeed	don't generate code
⊞ AfterNewSpeed	don't generate code
⊟ OnInterrupt	generate code
Event procedure name	TI1_OnInterrupt
Priority	same as interrupt

Writing an Event Handler

Event handler is a subroutine that is assigned to a specific event. After the event is enabled, Processor Expert generates the function with the specific name to the Event module. Refer to the Code Generation for details.

You can open the Event handler code (if it already exists) using a component pop-up menu **View/Edit event module** or double-click on the event. The event handler is an ordinary function and you need not to provide the interrupt handling specific code in the event code.

Interrupt Service Routines

When High or Low-level components are used, the interrupts functionality is covered by the events of the components. The interrupt subroutines calling user's event handlers are generated to the component modules and PE provides parts of the background code necessary to handle the interrupt requests correctly.

The Peripheral Initialization components can only provide the initialization of the interrupt and generate a record to the Interrupt Vector Table. You have to provide a full implementation of the interrupt subroutine. Refer to the Typical Usage of Peripheral Initialization Components for details.

Processor Expert Priority System

Some processors support selectable interrupts priorities. You may select a priority for each interrupt vector. The interrupt with a higher priority number can interrupt a service routine with the lower one.

Processor Expert supports the following settings in design-time: Interrupt Priority and priority of the event code. Priority can also be changed in the user code. You may use a processor component method to adjust the priority to a requested value.

Interrupt Priority

You may select interrupt priority in the component properties, just below the interrupt vector name. Processor Expert offers the following values, which are supported for all microcontrollers:

- minimum priority
- low priority
- medium priority
- high priority
- maximum priority

The selected value is automatically mapped to the priority supported by the target microcontroller. It is indicated in the third column of the **Component Inspector** view.

You may also select a target-specific numeric value (such as priority 255), if portability of the application to another architecture is not required.

Peripheral Initialization components on some platforms also allow to set the default value that means that you don't have any requirement, so the priority value will be the default after-reset value.

Priority of Event Code

Version Specific Information for Kinetis and ColdFire+ Derivatives

Priority of event code is not supported for Kinetis and ColdFire+.

You can also select a priority for the processing of the event code. This setting is available for the events that are invoked from the Interrupt Service Routines. This priority may be different from the interrupt priority. However, the meaning of the number is same, the event may be interrupted only by the interrupts with the higher priority. Processor Expert offers the following architecture independent values:

- same as interrupt — default value which means that Processor Expert does not generate any code affecting the priority of the event; the priority is in the state determined by the default hardware behavior.
- minimum priority
- low priority
- medium priority
- high priority
- maximum priority
- interrupts disabled — For example, the highest priority supported by the microcontroller, which may be interrupted only by non-maskable interrupts.

The selected value is automatically mapped to the priority supported by the target microcontroller and the selected value is displayed in the third column of the **Component Inspector**.

Refer to the version specific information below. You may also select a target-specific value, if portability of the application to another architecture is not required.

NOTE Some events do not support priorities because their invocation is not caused by the interrupt processing.

WARNING! Processor Expert does not allow you to decrease an event code priority (with the exception of 'Interrupts enabled' value on some platforms). This is because Processor Expert event routines are not generally reentrant so there is a risk that the interrupt would be able to interrupt itself during the processing. If there is such functionality requested, you have to do it manually (for example, by calling a appropriate processor component method setting a priority) and carefully check possible problems.

Version Specific Information for RS08 with Interrupt Support

Because of architecture limitations, the Processor Expert allows only interrupts disabled value so the interrupt is always disabled within the event routines. The same as interrupt value is mapped to interrupts disabled.

Configurations

You can have several configurations of the project in one project file. The configuration system is very simple. Every configuration keeps the enable/disable state of all components in the project (it does not keep any component settings). If you enable/disable a component in the project, the component state is updated in the currently selected configuration. If you create a new configuration the current project state is memorized.

Configurations of the current project are listed in the **Generator_Configurations** folder of the **Components** view.

Configurations can also hold additional settings that may influence code generation. These settings can be changed in the configuration inspector. Refer to the Configuration Inspector for details.

The symbol for conditional compilation is defined if it is supported by the selected language/compiler. The symbol **PEcfg_*[ConfigurationName]*** is defined in the processor interface.

You can switch using this symbol between variants of code according to the active configuration (see example in this chapter).

Configuration also stores which processor is selected as the target processor.

If the name of the configuration matches the name of one of the CodeWarrior's targets, the target is automatically selected as an active target when the user runs code generation.

NOTE It is possible to have two components with the same name in project. Each of the components could be enabled in different configuration. This way you can have different setup of a component (a component with the same name) in multiple configurations.

Example

Suppose, there is a configuration named, `Testing case`. You can use a component and part of our code using the component only in the `Testing case` configuration. Then you can make the testing case configuration active. After the successful code generation, the `Cpu.h` file contains the following definition:

```
/* Active configuration define symbol */

#define PEcfg_Testingcase 1
```

Add the following lines:

```
...

#ifdef PEcfg_TestingCase

Component_MethodCall(...);

#endif

...
```

Design Time Checking: Consequences and Benefits

During the design time, Processor Expert performs instant checking of the project. As a result of this checking, error messages may appear in the **Problems** view or directly in the third column of the **Component Inspector** (on the faulty items line). Sometimes, it may happen that only one small change in the project causes several (general) error messages.

On-Chip Peripherals

Some components use on-chip peripherals. In the **Component Inspector**, you can choose from all possible peripherals that can be used for implementation of the function of the current component. Processor Expert provides checking for required peripheral features such as word width and stop bit for serial channel, pull resistor for I/O pin and others.

Processor Expert also protects against the use of one peripheral in two components. If the peripheral is allocated for one component then the settings of this peripheral cannot be changed by any other component. The state of an allocated peripheral should never be changed directly in the user code. (Using special registers, I/O ports etc.) It is recommended to always use methods generated by Processor Expert. If the functionality of generated methods is not sufficient for your application, you can use PESL (Processor Expert System Library). Refer to the Low-level Access to Peripherals topic for details.

Note that if a peripheral is allocated to any component, all its parts are reserved. For example, if you use the 8-bit I/O port, all the I/O pins of the port are allocated and it is not possible to use them in other components.

In some timer components, you can choose if you want to use only a part of the timer (compare register) or an entire timer. If you select the entire timer, the driver can be optimized to work best with the timer. For example, invoke reset of the timer whenever it is needed by the component function.

Interrupt Priority

If the target processor shares interrupt priority between several interrupt vectors or shares interrupt vectors, Processor Expert provides checking of interrupt priority settings. For detailed information about Interrupt Priority, refer to the Interrupt Priority topic.

Memory

Processor Expert always checks the usage of the internal and external memories accessible throught processor address and data bus. Position and size of internal memory is defined by the processor type and can be configured in the processor Properties (if supported). External memories must be defined in processor Properties.

Any component can allocate a specific type of memory. Processor Expert provides checking of memory and protects you from making a wrong choice. For example, if a component requires external Flash, it is not possible to enter an address in internal RAM.

The bits can also allocate memory. Therefore, you can be sure that only one component uses an allocated bit of a register in external address space.

Timing

The settings of all timed high-level components are checked using the internal timing model. Refer to the Timing Settings topic for details. If there is no error reported, it means that Processor Expert was successful in calculating the initialization and runtime control values for all components and hence the settings should work according to the configuration.

Timing Settings

Many high-level components contain a timing configuration (for example, speed of the serial communication, period of the interrupt, conversion time of the ADC). Processor Expert allows to configure such timing using user-friendly units and it also contains a model of the complete microcontroller timing. This model allows calculation of the required values of control registers and continuous validation of the timing settings.

Timing Model

A component timing can be viewed like a chain of elements, such as dividers and multipliers between the main clock source and the device configured by the component. You can set the desired timing value using the **Timing** dialog box (refer to the Dialog Box for Timing Settings topic for details) or directly by specifying the value in **Component Inspector** (refer to the Syntax for the Timing Setup in the Component Inspector topic for details). Processor Expert tries to configure individual elements in the timing chain to achieve the result and the user is informed if it was successful. After leaving the **Timing** dialog box, the real value of the timing is shown in the third column of the component inspector.

Timing Setup Problems

The errors are reported in red in the **Timing** dialog box or in the timing property line in the **Component Inspector**. The error summary is available in the Error window. Follow the error message text to find the source of the problem. If no error is reported, it means that Processor Expert can achieve the desired timing for all components in the project.

Common problems that make impossible to set a timing value:

- It is impossible to set some item(s).

  This problem is reported in the component or the **Timing** dialog box and the user is informed which value has incorrect value. The reason is usually the hardware structure and limitations of the processor. The **Timing** dialog box shows the list of values (ranges) that are allowed to be set. It might be necessary to increase the allowed error (using the 'Error' field in the Timing dialog) that specifies the allowed difference between the required value and possible value that can be produced by the hardware.

- Settings for the device are mutually incompatible (or can't be used with another device).

  In this case, the problem is reported by all components that share some timing hardware. Due to dependencies between used parts of the timer, it is necessary to adjust the values of the shared elements (such as prescalers) to the same value.

  For example, if two `TimerInt` components are using two channels of one timer and all timer channels are connected to one common prescaler, it is not possible to set the values that would require a different prescaler values. In this case, it is useful to manually adjust the prescaler values of all components to the same value (switch to Expert view mode and adjust the property in the Component Inspector view).

- The Runtime setting from interval is selected and it is not possible to set the values.

  The required run-time settings are outside the range of one prescaler. This is a limitation of this mode of runtime setting.

Run-time Timing Settings Limitation

Some components allow to change the timing at run-time by switching among several predefined values or by setting a value from given interval.

For the runtime setting from interval the prescaler value is fixed and the Processor Expert allows to adjust the time using a compare/reload registers. It means that Processor Expert allows to configure the limits of an interval only within a range of one prescaler and it is possible to set values from this interval only. Refer to the Dialog Box for Timing Settings topic for details.

Speed Modes

Processor Expert provides three speed modes that are generalization of all the possible processor clock speed modes used for power-saving supported by most of the modern microcontrollers. Refer to the Speed Modes Support topic for details.

Creating User Component Templates

If you frequently use a component with the same specific settings, you may need to save the component with its settings as a template. This template is displayed in the Components Library View view under given name, behaves as a normal component and could be added to any project. The template has the same properties as the original component. The value of the properties are preset in the template and could be marked as read only.

This section describes how to create a component template and save it.

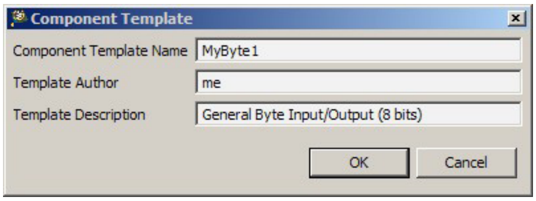
Creating and Saving Templates

1. Open the pop-up menu of the component in the **Component** view and select **Save component settings as template**.

   Alternatively, you can use the Component Inspector window: open the view menu using the arrow icon in the top right corner and select Save component settings as template.

2. Fill in the template details into the dialog box and confirm:

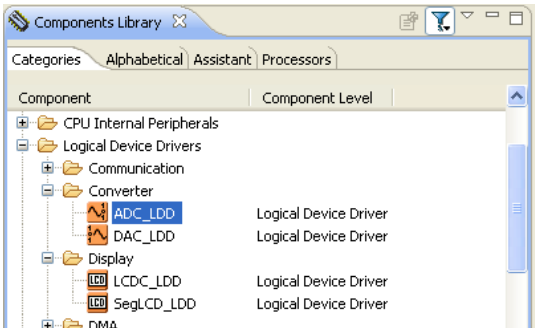
Figure 3.11 Component Template



Component Template Name	MyByte1
Template Author	me
Template Description	General Byte Input/Output (8 bits)

3. The template appears within the Components Library View and can be inserted into projects. It may be necessary to invoke refresh command by selecting the pop-up menu of **Components Library** and select **Refresh** option.

Figure 3.12 Components Library View



Component	Component Level
CPU Internal Peripherals	
Logical Device Drivers	
Communication	
Converter	
ADC_LDD	Logical Device Driver
DAC_LDD	Logical Device Driver
Display	
LCDC_LDD	Logical Device Driver
SegLCD_LDD	Logical Device Driver
DMA	

Signal Names

The main purpose of signals allows you to name the pins used by components with names corresponding to the application.

Assigning Signals to Pins

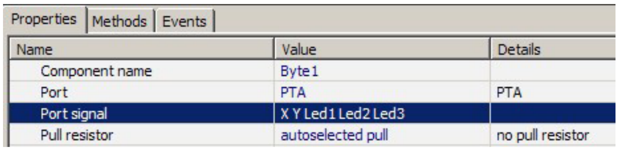
A signal name can be assigned to an allocated pin by specifying the signal name into the appropriate property (for example, `Pin_signal`) in the Component Properties (available in Advanced view mode). Signal name is an identifier that must start with a letter and rest of the name must contain only letters, numbers, and underscore characters.

For the components that allocate a whole port, such as ByteIO, there are two options:

- Assign a same signal name to all pins of port by writing the name into the **Port signal** property. Processor Expert automatically assigns this name extended with a bit number suffix to each of the individual pins.

- Assign a different signal names to individual pins by writing pin signal names (from the lowest bit to the highest one) separated by commas or spaces into the **Port signal** property.

Figure 3.13 Signal Names List for a Port



Name	Value	Details
Component name	Byte1	
Port	PTA	PTA
Port signal	X Y Led1 Led2 Led3	
Pull resistor	autoselected pull	no pull resistor

Generated Documenation

Processor Expert automatically generates a document `{projectname}_SIGNALS.txt` or `{projectname}_SIGNALS.doc` containing a list of relationship between defined signals and corresponding pins. There is an additional signal direction information added next to each signal name and pin number information next to each pin name. This document can be found in the `Documentation` folder of the **Component** view.

Listing 3.1 Sample of Generated Signals Documentation

```
=================================================================
SIGNAL LIST
-----------------------------------------------------------------
SIGNAL-NAME [DIR] => PIN-NAME [PIN-NUMBER]
-----------------------------------------------------------------
LED1 [Output] => GPIOA8_A0 [138]
LED2 [Output] => GPIOA9_A1 [10]
Sensor [Input] => GPIOC5_TA1_PHASEB0 [140]
TestPin [I/O] => GPIOE0_TxD0 [4]
Timer [Output] => GPIOC4_TA0_PHASEA0 [139]
=================================================================
=================================================================
PIN LIST
-----------------------------------------------------------------
PIN-NAME [PIN-NUM] => SIGNAL-NAME [DIRECTION]
-----------------------------------------------------------------
GPIOA8_A0 [138] => LED1 [Output]
GPIOA9_A1 [10] => LED2 [Output]
GPIOC4_TA0_PHASEA0 [139] => Timer [Output]
GPIOC5_TA1_PHASEB0 [140] => Sensor [Input]
GPIOE0_TxD0 [4] => TestPin [I/O]
=================================================================
```

Component Inheritance and Component Sharing

Basic Terms

- **Ancestor** is a component that is inherited (used) by another component.
- **Descendant** is a new component that inherits (uses) another component(s).
- **Shared Ancestor** is a component that can be used and shared by multiple components.

Inheritance

Inheritance in Processor Expert means that an ancestor component is used only by the descendant component. Inheritance is supported in order to allow components to access peripherals by hardware-independent interface of the ancestor components. For example, a component that emulates a simple I2C transmitter may inherit two BitIO components for generation of an output signal.

On several complex components inheritance is used to separate component settings into several logical parts, for example, settings of channel is inherited in the component with settings of the main peripheral module.

Settings in Processor Expert

The Descendant component contains a property that allows selecting an ancestor component from a predefined list of templates. The component is created after selection of an appropriate template name (or component name) from the list of the templates fitting the specified interface. Any previously used ancestor component is discarded.

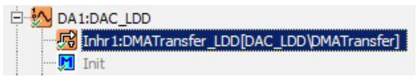
Figure 3.14 Inherited Component Item in Inspector

DMATransfer	DMATransfer	

Processor Expert allows you to select from ancestors that implement the required interface and are registered by the descendant component.

The ancestor component is displayed under its descendant in the project structure tree in the **Components** view.

Figure 3.15 Example of Ancestor and Descendant Components in the Components View



```
DA1:DAC_LDD
    Inhr1:DMATransfer_LDD[DAC_LDD\DMATransfer]
    Init
```

An ancestor component requires a list of methods and events (interface), which must be implemented by an ancestor component. The error is shown if the ancestor component does not implement any of them. For example, if the settings of the descendant component do not allow it to generate this method.

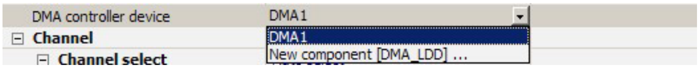
Component Sharing

Component sharing allows you to cause several components to use capability of one component similar to inheritance. This feature allows sharing of its resources and its drivers with other components. For example, components may share an I2C component for communication with peripherals connected to the I2C bus or some component may do DMA transfers using DMA component.

Settings in Processor Expert

A shared ancestor component contains a property that allows you to select existing shared ancestor component or create a new one. In this case, the ancestor component is included in the project tree as the other components. The ancestor component may be used with the descendant component only if it is created from a template registered in the descendant component or if the component type is registered in the descendant component. It is recommended that you always create a shared ancestor component through a descendant component.

Figure 3.16 Popup Menu for Selection/Creation of a Shared Ancestor Component


DMA controller device DMA1
Channel
Channel select

Run-time Resources Allocation

Processor Expert (generated code) does not check the usage of shared resources/code. It's up to you to ensure the correct run-time resources allocation of a shared ancestor component. Often, it is not possible for a shared ancestor component to be used simultaneously by several components.

Pin Sharing

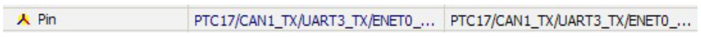
Sharing Pins Among Peripherals

Some processors allows few pins to be used by multiple peripherals. This may lead to the need of sharing pin(s) by multiple components. Normally, if you select one pin in more than one component, a conflict is reported. However, it is possible to setup a sharing for such pin in the component inspector.

One of the components sharing a pin has to be chosen as a main component. This component will initialize the pin. In the properties of other components that use the pin, the pin has to be marked as shared (see figure below).

Pin sharing can be set in the **Component Inspector**. The **Component Inspector** must be in **Expert** view mode. Use the pop-up menu of the property and select the command **Pin Sharing Enabled**.

Figure 3.17 Pin Property with Sharing Enabled


Pin PTC17/CAN1_TX/UART3_TX/ENET0_... PTC17/CAN1_TX/UART3_TX/ENET0_...

Pin sharing is advanced usage of the processor peripherals and should be done only by skilled users. Pin sharing allows advanced usage of the pins even on small processor packages and allows application-specific usage of the pins.

ConnectPin Method

It is necessary to invoke the component method `ConnectPin` to connect a component to the shared pin. It is also necessary to invoke the main component method to connect pin back to the main component. In fact, the peripherals can usually operate simultaneously, but they have no connection to the shared pins unless the *`ConnectPin`* method is executed. In case that all components control the shared pin using one peripheral, it is not necessary to use the *`ConnectPin`* method.

Shared pins are presented in the Processor View as well. The component to pin connection line is red.

Export and Import

Processor Expert allows to import or export component settings or configuration of selected Processor Expert components.

This topic explains:

- Export Component Settings
- Export Board Configuration
- Apply Board Configuration
- Component Settings to Project
- Component(s) to Components Library

Export Component Settings

It is possible to export one or more component settings, such as:

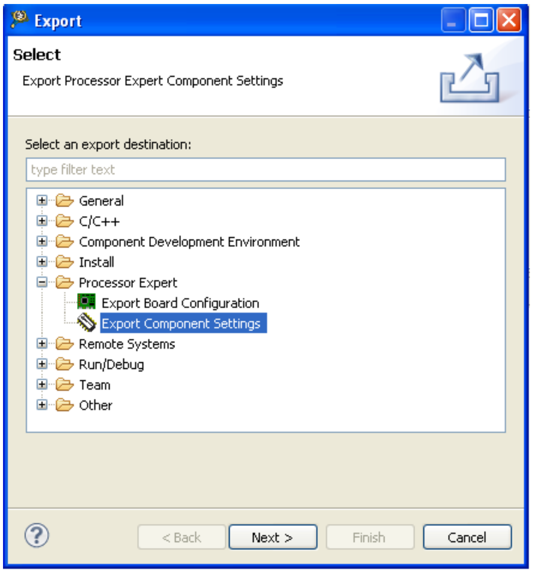
- Configurations
- Operating system
- Processors
- Components

To export component settings:

1. In the IDE, select **File > Export**. The **Export** wizard appears.

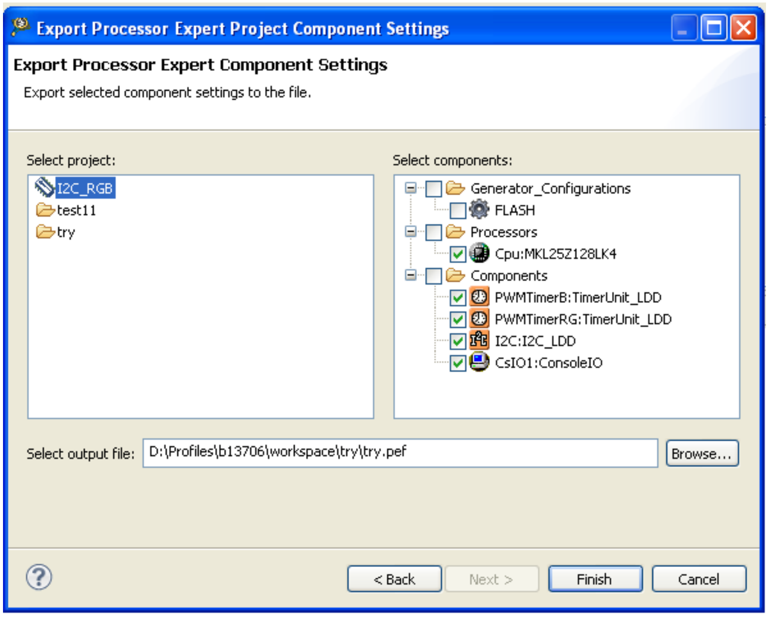
   Expand **Processor Expert** tree. Select **Export Component Settings** option as shown below.

Figure 3.18 Export Wizard


Export
Select
Export Processor Expert Component Settings
Select an export destination:
type filter text
General
C/C++
Component Development Environment
Install
Processor Expert
Export Board Configuration
Export Component Settings
Remote Systems
Run/Debug
Team
Other
< Back
Next >
Finish
Cancel

2. Click **Next**. The **Export Processor Expert Component Settings** page appears.

Figure 3.19 Export Processor Expert Component Settings Page



In the left panel, select the project for which you want to export the component settings. In the right panel, select the components to export. Click the **Browse** button to select the output file in which the export settings are saved. The default location for saving the output file is recently selected folder, your home directory. The extension of the file is `.pef`.

3. Click **Finish** to complete the exporting of component settings.

Export Board Configuration

It is possible to export one processor and one or more components (the components automatically selected are CPU, Pin settings, LDD, and Init components)

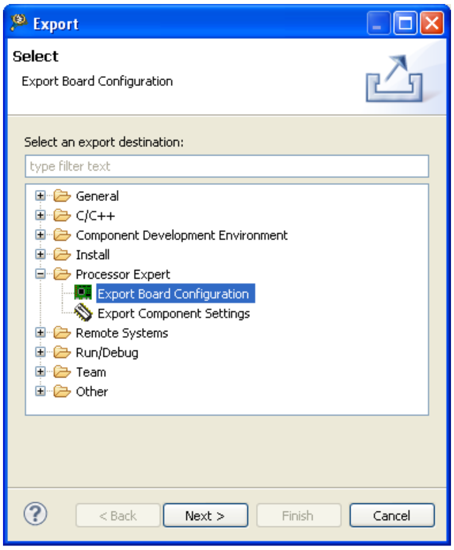
You can save the current state of components related to board configuration in to the external file. The extension of the file is `.peb`. The default location for saving the output file is recently selected folder, your home directory.

To export board configuration:

1. In the IDE, select **File > Export**. The **Export** wizard appears.

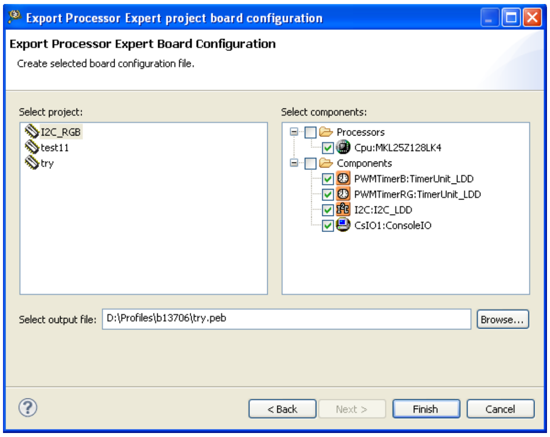
   Expand **Processor Expert** tree. Select **Export Board Configuration** option as shown below.

Figure 3.20 Export Wizard


Export
Select
Export Board Configuration
Select an export destination:
type filter text
- General
- C/C++
- Component Development Environment
- Install
- Processor Expert
  - Export Board Configuration
  - Export Component Settings
- Remote Systems
- Run/Debug
- Team
- Other

2. Click **Next**. The **Expert Processor Expert Board Configuration** page appears.

Figure 3.21 Expert Processor Expert Board Configuration Page


Export Processor Expert project board configuration
Export Processor Expert Board Configuration
Create selected board configuration file.
Select project:
- I2C_RGB
- test11
- try

Select components:
- Processors
  - Cpu:MKL25Z128LK4
- Components
  - PWMTimerB:TimerUnit_LDD
  - PWMTimerRG:TimerUnit_LDD
  - I2C:I2C_LDD
  - CsIO1:ConsoleIO

Select output file: D:\Profiles\b13706\try.peb

In the left panel, select the project for which you want to export the board settings. In the right panel, the processor and components are already selected. Click the **Browse** button to select the output file in which the export settings are saved. The default location for saving the output file is either recently selected folder or your home directory. The extension of the file is `.peb`.

3. Click **Finish** to complete the exporting of board configurations.

Apply Board Configuration

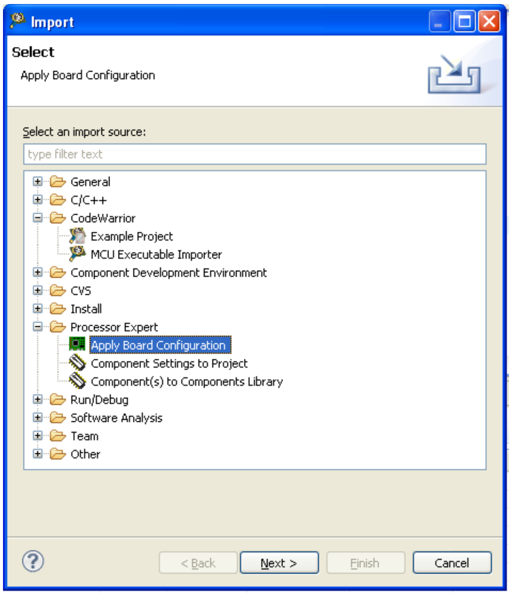
You can import board configuration from a file. The imported configurations are added into the selected project.

To import board configuration:

1. In the IDE, select **File > Import**. The **Import** wizard appears.

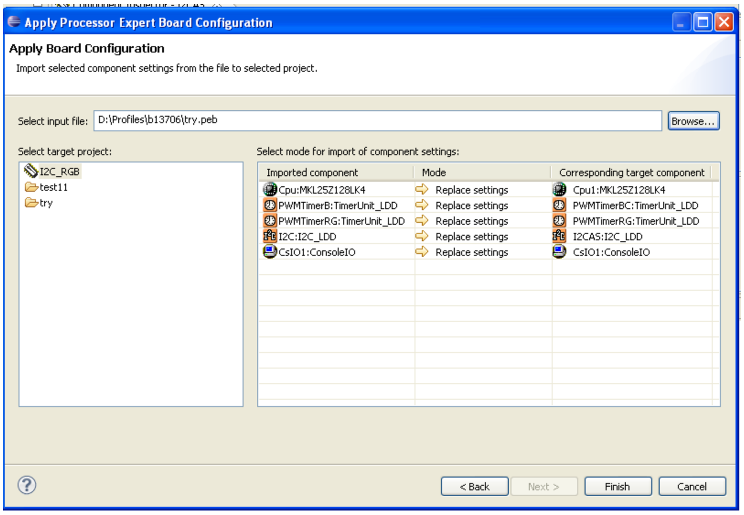
   Expand **Processor Expert** tree. Select **Apply Board Configuration** option as shown below.

Figure 3.22 Import Wizard



2. Click **Next**. The **Apply Board Configuration** page appears.

Figure 3.23 Apply Board Configuration Page



Apply Processor Expert Board Configuration

Apply Board Configuration

Import selected component settings from the file to selected project.

Select input file: D:\Profiles\b13706\try.peb

Select target project:

- I2C_RGB
- test11
- try

Select mode for import of component settings:

Imported component	Mode	Corresponding target component
Cpu:MKL25Z128LK4	Replace settings	Cpu1:MKL25Z128LK4
PWMTimerB:TimerUnit_LDD	Replace settings	PWMTimerBC:TimerUnit_LDD
PWMTimerRG:TimerUnit_LDD	Replace settings	PWMTimerRG:TimerUnit_LDD
I2C:I2C_LDD	Replace settings	I2CAS:I2C_LDD
CsIO1:ConsoleIO	Replace settings	CsIO1:ConsoleIO

Before importing, you can rename some of the components (as shown in figure above), so it will show the mapping of components with different names, but same device allocation.

To import component settings from the file to selected project, click the **Browse** button. Select the input file with the `.peb` extension. The default option **Replace settings** is selected if imported component settings are having same peripheral device allocation (or ID) and type.

For more information on different types of modes, refer to the Component Settings to Project topic.

3. Click **Finish**. The settings from the `.peb` file is imported to the selected project.

Component Settings to Project

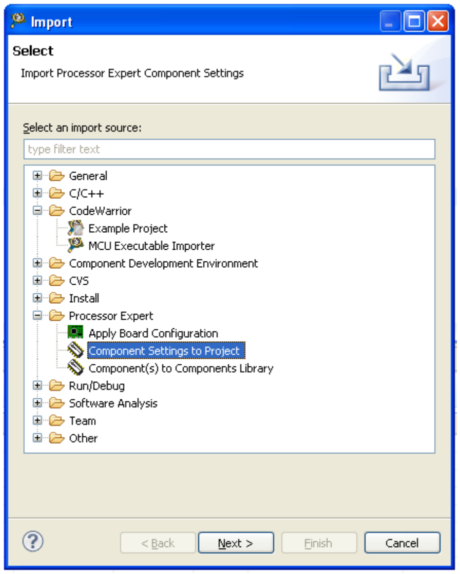
You can import one or more component settings from a file. Although, components with existing name results in conflict, but it is still possible to import. The imported components are added into the selected project.

To import component settings:

1. In the IDE, select **File > Import**. The **Import** wizard appears.

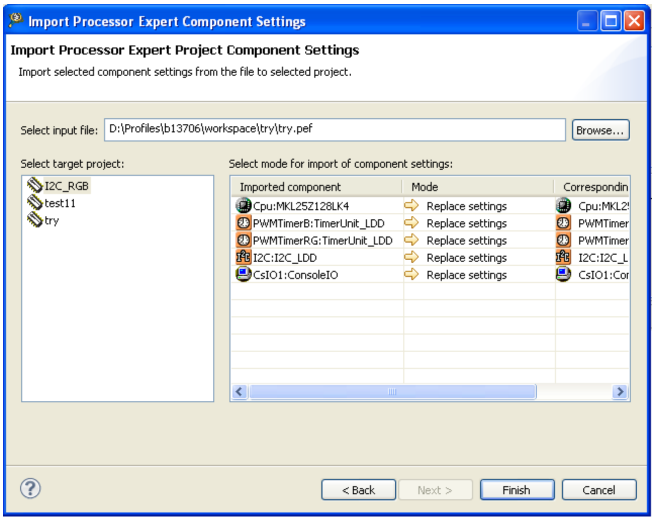
   Expand **Processor Expert** tree. Select **Component Settings to Project** option as shown below.

Figure 3.24 Import Wizard



2. Click **Next**. The **Import Component Settings** page appears.

Figure 3.25 Import Component Settings Page



Import Processor Expert Component Settings

Import Processor Expert Project Component Settings

Import selected component settings from the file to selected project.

Select input file: D:\Profiles\b13706\workspace\try\try.pef

Select target project: I2C_RGB, test11, try

Select mode for import of component settings:

Imported component	Mode	Correspondin
Cpu:MKL25Z128LK4	Replace settings	Cpu:MKL2!
PWMTimerB:TimerUnit_LDD	Replace settings	PWMTimer
PWMTimerRG:TimerUnit_LDD	Replace settings	PWMTimer
I2C:I2C_LDD	Replace settings	I2C:I2C_L
CsIO1:ConsoleIO	Replace settings	CsIO1:Cor

To import component settings from the file to selected project, click the **Browse** button. Select the input file with the `.pef` or `.peb` or `.pe` extension. The default option **Replace settings** is selected if imported component settings are having same name (or ID) and type.

You can select the mode for importing components settings, the options are:

- **Ignore** — do not import this component settings
- **Add new** — add new component with imported settings
- **Add new, keep existing** — if component with same name or type exists, it will add a new component with imported settings and keep the existing one (may cause conflicts)
- **Add new, disable existing** — if component with same name or type exists, it will add a new component with imported settings and disable the existing one
- **Replace settings** — replace existing component with new settings from imported file

3. Click **Finish**. The settings from the `.pef` file is imported to the selected project.

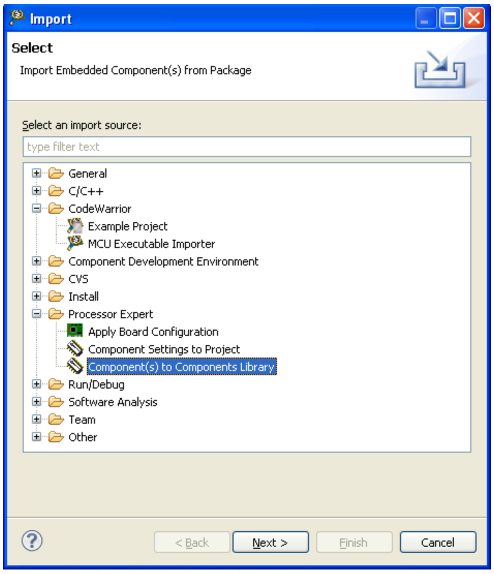
Component(s) to Components Library

To import a component:

1. In the IDE, select **File > Import**. The **Import** wizard appears.

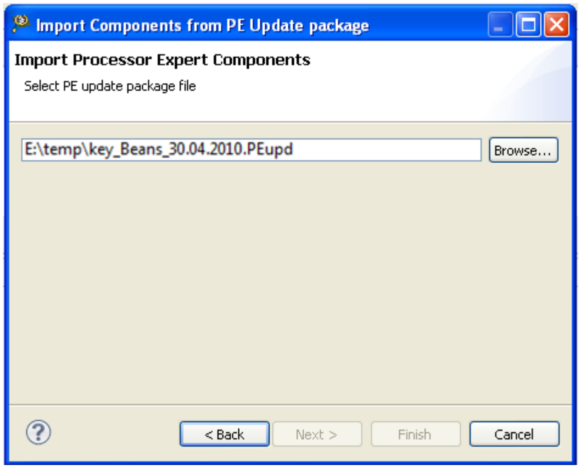
Expand **Processor Expert** tree. Select **Component(s) to Component Library** option as shown below.

Figure 3.26 Import Wizard



2. Click **Next**. The Import Processor Expert Components page appears.

Figure 3.27 Import Processor Expert Components Page


Import Components from PE Update package
Import Processor Expert Components
Select PE update package file
E:\temp\key_Beans_30.04.2010.PEupd
Browse...
< Back
Next >
Finish
Cancel

3. Click **Finish** to select and install Processor Expert update packages (.PEUpd) files.

Implementation Details

This topic explains implementation details for Embedded Components and Processor Expert generated code.

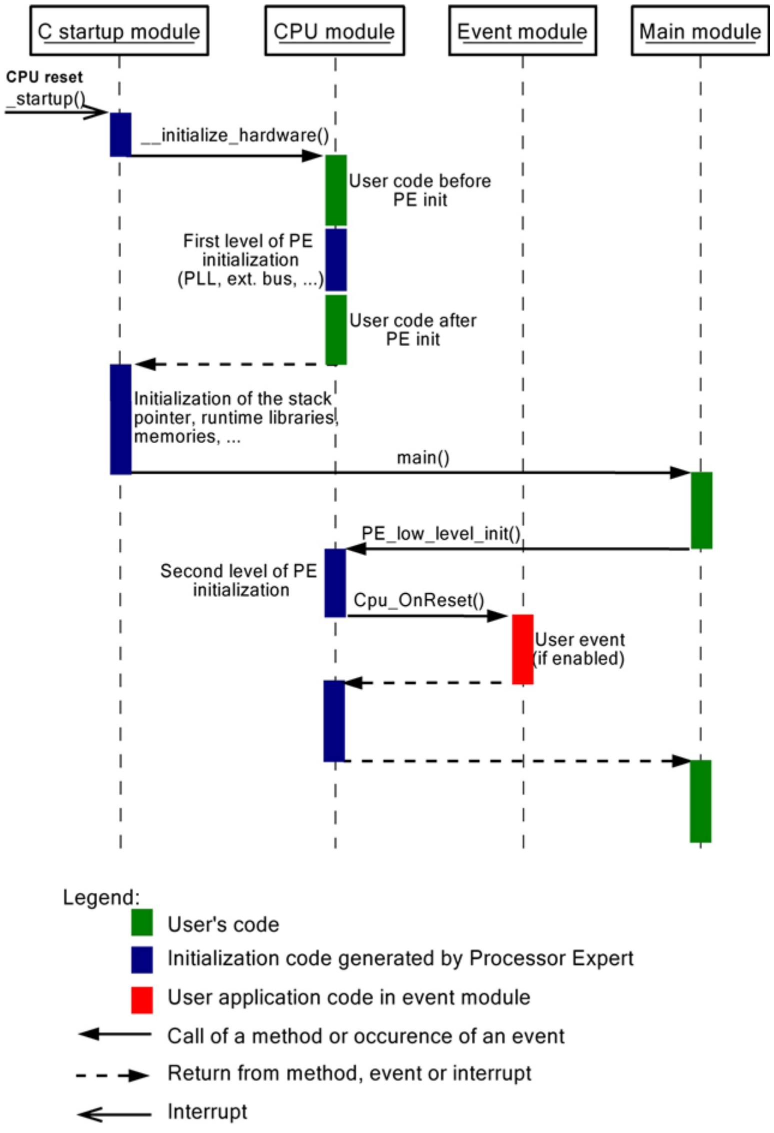
The following describes:

- Reset Scenario with PE for ColdFire and Kinetis Microcontrollers
- Version Specific Information for Kinetis and ColdFire+

Additional implementation specific information can be found on individual component documentation pages.

Reset Scenario with PE for ColdFire and Kinetis Microcontrollers

Figure 3.28 Reset Sequence Diagram with Processor Expert


C startup module
CPU module
Event module
Main module
CPU reset
_startup()
__initialize_hardware()
User code before PE init
First level of PE initialization (PLL, ext. bus, ...)
User code after PE init
Initialization of the stack pointer, runtime libraries, memories, ...
main()
PE_low_level_init()
Second level of PE initialization
Cpu_OnReset()
User event (if enabled)
Legend:
User's code
Initialization code generated by Processor Expert
User application code in event module
Call of a method or occurence of an event
Return from method, event or interrupt
Interrupt

_startup()

The `_startup()` function is called as the first function after the reset. The `_startup()` function initializes the stack pointer, calls the `__initialize_hardware()` function and continues with initialization of the enviroment (such as memory initialization). At the end of the `_startup()` function the `main()` function is called.

__initialize_hardware()

The `__initialize_hardware()` function is called from the `_startup` function after an initialization of the stack pointer. This function is defined in the cpu module, usually `Cpu.c`, and provides necessary system initialization such as PLL, and external bus.

Sometimes it is necessary to do some special user initialization immediately after the cpu reset. Processor Expert provides a possibility to insert user code into the `__initialize_hardware()` function. There is a User Initialization property in the build options tab of a processor component inspector defined for this purpose. Refer to the Component Inspector topic for details.

PE_low_level_init()

There is a second level of Processor Expert initialization `PE_low_level_init()` called at the beginning of the `main()` function. `PE_low_level_init()` function provides initialization of all components in project and it is necessary for proper functionality of the Processor Expert project.

OnReset Event

You can write the code that will be invoked from the `PE_low_level_init()` function after Processor Expert internal initialization, but before the initialization of individual components. Thus, you should expect that peripherals are not completely initialized yet. This event can be enabled/disabled in the processor component inspector's events page.

Version Specific Information for Kinetis and ColdFire+

Only the Peripheral Initialization and Logical Device Drivers (LDD) components are available for the Kinetis and ColdFire+ derivatives. For details, refer to the Logical Device Drivers and Component Categories topics.

Kinetis and ColdFire+ processor components support Clock configurations that are similar to Speed Modes (available with High Level components) but provide more options on configuring low power and slow clock modes of the processor. Refer to the details on individual settings in the processor component's on-line help.

Code Generation and Usage

It expalis you about the principles and results of the Processor Expert code generation process and the correct ways and possibilities of using this code in the user application.

Refer to the following topics for more information:

- Code Generation
- Predefined Types, Macros and Constants
- Typical Usage of Component in User Code
- User Changes in Generated Code

Code Generation

ProcessorExpert.pe pop-up menu > Generate Processor Expert Code Generate Code command initiates the code generation process. During this process source code modules containing functionality of the components contained in the project are generated. The project must be set-up correctly for successful code generation. If the generation is error-free all generated source code files are saved to the destination directory.

Files Produced by Processor Expert

The existence of the files can be conditional to project or Processor Expert environment settings and their usage by the components. Refer to the Project Static Modules topic for more details on files produced for project with static code support.

- **Component module**

  This module with its header file is generated for every component in the project with exception of some components that generate only an initialization code or special source code modules. Name of this file is the same as the name of the component.

  Header file (.h) contains definitions of all public symbols, which are implemented in the component module and can be used in the user modules.

  The module contains implementation of all enabled methods and may also contain some subroutines for internal usage only.

- **Processor module**

The processor module is generated according to the currently active target processor component. The processor module additionally contains:

– microcontroller initialization code

– interrupt processing

- **Main module**

  The main module is generated only if it does not already exist (if it exists it is not changed). Name of this module is the same as the name of the project.

  The main module contains the `main` function, which is called after initialization of the microcontroller (from the processor module). By default, this function is generated empty (without any reasonable code). It is designed so that you can write code here.

- **Event module**

  The event module is generated only if it does not exist. If it exists, only new events are added into the module; user written code is not changed.

  The event module contains all events selected in the components. By default, these event handler routines are generated empty (without any meaningful code). It is considered that user will write code here.

  Event module can also contain the generated ISRs for the components that require a direct interrupt handling (Peripheral Initialization Components). It is possible to configure the name of event module individually for each component in the ADVANCED view mode of the Component Inspector. However, note that the event module is not generated by Processor Expert if there is no event enabled in the component, except the processor component, for which the event module is always generated.

- Method list file with description of all components, methods and events generated from your project. The name of the file is `{projectname}.txt` or `{projectname}.doc`. This documentation can be found in the Documentation folder.

- **Signal names**

  This is a simple text file `{projectname}_SIGNALS.txt` or `{projectname}_SIGNALS.doc` with a list of all used signal names. The signal name can be assigned to an allocated pin in the component properties (available in ADVANCED view mode). This documentation can be found in the Documentation folder of the **Components** view. Refer to the Signal Names topic for details.

- **Code generation log** that contains information on changes since last code generation. Refer to the Tracking Changes in Generated Code for details.

- **XML documentation** containing the project information and settings of all components in XML format. The generated file `{projectname}_Settings.xml` can be found in the Documentation folder of the **Components** view. It is updated after each successful code generation.

- **Shared modules** with shared code (the code which is called from several components). Complete list of generated shared modules depends on selected processor, language, compiler and on the current configuration of your project. Typical shared modules are:
  - **IO_Map.h**

    Control registers and bit structures names and types definitions in C language.
  - **IO_Map.c**

    Control registers variable declarations in C language. This file is generated only for the HC(S)08/HC(S)12 versions.
  - **Vectors.c**

    A source code of the interrupt vector table content.
  - **PE_Const.h**

    Definition of the constants, such as speed modes, reset reasons. This file is included in every driver of the component.
  - **PE_Types.h**

    Definition of the C types, such as bool, byte, word. This file is included in every driver of the component.
  - **PE_Error.h**

    Common error codes. This file contains definition of return error codes of component's methods. See the generated module for detailed description of the error codes. This file is included in every driver of the component.
  - **PE_Timer**

    This file contains shared procedures for runtime support of calculations of timing constants.
  - **{startupfile}.c**

    This external module, visible in the External Modules folder of the **Components** view, contains a platform specific startup code and is linked to the application. The name of the file is different for the Processor Expert versions.
  - **“PESL".h**

    PESL include file. This file can be included by the user in his/her application to use the PESL library. For more details, refer to the Processor Expert System Library topic.
- Static modules with static source code of Peripheral Initialization components, processor component methods including startup methods, IO map, Physical Device Driver macros and other Processor Expert modules. These modules are available

only for projects with static code support. For more details, refer to the Project Static Modules topic.

- Configuration header files containing parameterization constants which controls static source code. Configuration header file is generated for each component with static code support added to the project. These modules are generated only for projects with static code support. For more details, refer to the Peripheral Initialization Component Files topic.

For more details, refer to the Predefined Types, Macros and Constants topic.

Tracking Changes in Generated Code

Processor Expert allows to track changes in generated modules. It is just necessary to enable the option **Create code generation log** in the Processor Expert Project options. Refer to the Processor Expert Options topic for details. If this option is enabled, a file `ProcessorExpert_CodeGeneration.txt` is generated into Documentation folder.

The file contains a list of changes with details on purpose of each change. Refer to the example below:

Listing 3.2 Example — Tracking Changes in Generated Code

```
###############################################################
Code generation 2010/10/22, 15:57; CodeGen: 1 by user:
by Processor Expert 5.00 for Freescale Microcontrollers; PE core 04.46
Configuration: Debug_S08GW64CLH
Target CPU: MC9S08GW64_64; CPUDB ver 3.00.000
# The following code generation options were changed:
> option Create code generation log: value changed from false to true
###############################################################
Code generation 2010/10/22, 16:01; CodeGen: 2 by user: hradsky
by Processor Expert 5.00 Beta for Freescale Microcontrollers; PE core
04.46
Configuration: Debug_S08GW64CLH
Target CPU: MC9S08GW64_64; CPUDB ver 3.00.000
# Component Cpu:MC9S08GW64_64, the following files modified due to
internal interdependency:
- Generated_Code\Vectors.c - changed
- Generated_Code\Cpu.h - changed
- Generated_Code\Cpu.c - changed
# New component PWM1:PWM (ver: 02.231, driver ver. 01.28) added to the
project, the following - 78 -
Processor Expert User Manual Application Design
- Generated_Code\PWM1.h - added
- Generated_Code\PWM1.c - added
# Documentation
```

```
- Documentation\ProcessorExpert.txt - regenerated
- Documentation\ProcessorExpert_Settings.xml - regenerated
# Other files have been modified due to internal interdependency:
- Generated_Code\PE_Timer.h - added
- Generated_Code\PE_Timer.c - added
# User modules
- Sources\ProcessorExpert.c - changed
> updated list of included header files
- Sources\Events.h - changed
> updated list of included header files
Totally 11 file(s) changed during code generation.
```

To view changes within the individual files, you can use a file pop-up menu command **Compare with > Local history...** available in Components view. It allows to compare files with the version before the code generation.

Predefined Types, Macros and Constants

Processor Expert generates definitions of all hardware register structures to the file `IO_Map.h`. The Processor Expert type definitions are generated to the file `PE_Types.h` which also containins definitions of macros used for a peripheral register access. Refer to the Direct Access to Peripheral Registers topic for details.

Types

The following table lists the predefined types and their description:

Table 3.3 Predefined Types

Type	Description	Supported for
byte	8-bit unsigned integer (unsigned char)	all
bool	Boolean value (unsigned char) (TRUE = any non-zero value / FALSE = 0)	all
word	16-bit unsigned integer (unsigned int)	all
dword	32-bit unsigned integer (unsigned long)	all
dlong	array of two 32-bit unsigned integers (unsigned long)	all
TPE_ErrCode	Error code (uint8_t)	all except MPC55xx

Structure for Images

```
typedef struct { /* Image */
word width; /* Image width in pixels */
word height; /* Image height in pixels */
byte *pixmap; /* Image pixel bitmap */
word size; /* Image size in bytes */
char *name; /* Image name */
} TIMAGE;
typedef TIMAGE* PIMAGE ; /* Pointer to image */
```

Structure for 16-bit Register:

```
/* 16-bit register (big endian format) */
typedef union {
word w;
struct {
byte high,low;
} b;
} TWREG;
```

Version Specific Information for 56800/E

For information on SDK types definitions, go to the page SDK types.

Macros

```
__DI()            - Disable global interrupts
__EI()            - Enable global interrupts
EnterCritical()   - It saves CCR register and disable
                    global interrupts
ExitCritical()    - It restores CCR register saved in
                    EnterCritical()
```

For the list of macros available for Peripheral registers access, refer to the Direct Access to Peripheral Registers topic.

Constants

Methods Error Codes

The error codes are defined in the PE_Error module. Error code value is 8-bit unsigned byte. Range 0 - 127 is reserved for PE, and 128 - 255 for user.

ERR_OK	0	OK
ERR_SPEED	1	This device does not work in the active speed mode
ERR_RANGE	2	Parameter out of range
ERR_VALUE	3	Parameter of incorrect value
ERR_OVERFLOW	4	Timer overflow
ERR_MATH	5	Overflow during evaluation
ERR_ENABLED	6	Device is enabled
ERR_DISABLED	7	Device is disabled
ERR_BUSY	8	Device is busy
ERR_NOTAVAIL	9	Requested value not available
ERR_RXEMPTY	10	No data in receiver
ERR_TXFULL	11	Transmitter is full
ERR_BUSOFF	12	Bus not available
ERR_OVERRUN	13	Overrun is present
ERR_FRAMING	14	Framing error is detected
ERR_PARITY	15	Parity error is detected
ERR_NOISE	16	Noise error is detected
ERR_IDLE	17	Idle error is detected
ERR_FAULT	18	Fault error is detected

ERR_BREAK	19	Break char is received during communication
ERR_CRC	20	CRC error is detected
ERR_ARBITR	21	A node loses arbitration. This error occurs if two nodes start transmission at the same time
ERR_PROTECT	22	Protection error is detected
ERR_UNDERFLOW	23	Underflow error is detected
ERR_UNDERRUN	24	Underrun error is detected
ERR_COMMON	25	General unspecified error of a device. The user can get a specific error code using the method GetError
ERR_LINSYNC	26	LIN synchronization error is detected
ERR_FAILED	27	Requested functionality or process failed
ERR_QFULL	28	Queue is full

Version Specific Information for 56800/E

For information on SDK constants definitions, go to the page SDK types.

56800/E Additional Types For SDK Components

The following types definitions are generated into the file `PETypes.h` in the Processor Expert for 56800/E. These types are intended to be used with the algorithms coming from the original SDK library. For more details, refer to the appropriate components documentation.

Listing 3.3 56800/E Additional Types For SDK Components

```
/* SDK types definition */
typedef signed char Word8;
typedef unsigned char UWord8;
typedef short Word16;
typedef unsigned short UWord16;
typedef long Word32;
typedef unsigned long UWord32;
typedef signed char Int8;
typedef unsigned char UInt8;
typedef int Int16;
typedef unsigned int UInt16;
typedef long Int32;
typedef unsigned long UInt32;
typedef union
{
struct
{
UWord16 LSBpart;
Word16 MSBpart;
} RegParts;
Word32 Reg32bit;
} decoder_uReg32bit;
typedef struct
{
union { Word16 PositionDifferenceHoldReg;
Word16 posdh; };
union { Word16 RevolutionHoldReg;
Word16 revh; };
union { decoder_uReg32bit PositionHoldReg;
Word32 posh; };
}decoder_sState;
typedef struct
{
UWord16 EncPulses;
UWord16 RevolutionScale;
Int16 scaleDiffPosCoef;
UInt16 scalePosCoef;
Int16 normDiffPosCoef;
Int16 normPosCoef;
}decoder_sEncScale;
typedef struct
{
UWord16 Index :1;
UWord16 PhaseB :1;
UWord16 PhaseA :1;
UWord16 Reserved :13;
}decoder_sEncSignals;
typedef union{
decoder_sEncSignals EncSignals;
UWord16 Value;
} decoder_uEncSignals;
/
**********************************************************************
********
*
* This Motor Control section contains generally useful and generic
* types that are used throughout the domain of motor control.
*
```

```
**********************************************************************
********/
/* Fractional data types for portability */
typedef short Frac16;
typedef long Frac32;
typedef enum
{
mcPhaseA,
mcPhaseB,
mcPhaseC
} mc_ePhaseType;
typedef struct
{
Frac16 PhaseA;
Frac16 PhaseB;
Frac16 PhaseC;
} mc_s3PhaseSystem;
/* general types, primary used in FOC */
typedef struct
{
Frac16 alpha;
Frac16 beta;
} mc_sPhase;
typedef struct
{
Frac16 sine;
Frac16 cosine;
} mc_sAngle;
typedef struct
{
Frac16 d_axis;
Frac16 q_axis;
} mc_sDQsystem;
typedef struct
{
Frac16 psi_Rd;
Frac16 omega_field;
Frac16 i_Sd;
Frac16 i_Sq;
} mc_sDQEstabl;
typedef UWord16 mc_tPWMSignalMask;
/* pwm_tSignalMask contains six control bits
representing six PWM signals, shown below.
The bits can be combined in a numerical value
that represents the union of the appropriate
bits. For example, the value 0x15 indicates
that PWM signals 0, 2, and 4 are set.
*/
```

```
/* general types, primary used in PI, PID and other controllers */
typedef struct
{
Word16 ProportionalGain;
Word16 ProportionalGainScale;
Word16 IntegralGain;
Word16 IntegralGainScale;
Word16 DerivativeGain;
Word16 DerivativeGainScale;
Word16 PositivePIDLimit;
Word16 NegativePIDLimit;
Word16 IntegralPortionK_1;
Word16 InputErrorK_1;
}mc_sPIDparams;
typedef struct
{
Word16 ProportionalGain;
Word16 ProportionalGainScale;
Word16 IntegralGain;
Word16 IntegralGainScale;
Word16 PositivePILimit;
Word16 NegativePILimit;
Word16 IntegralPortionK_1;
}mc_sPIparams;
#endif /* __PE_Types_H */
#define MC_PWM_SIGNAL_0 0x0001
#define MC_PWM_SIGNAL_1 0x0002
#define MC_PWM_SIGNAL_2 0x0004
#define MC_PWM_SIGNAL_3 0x0008
#define MC_PWM_SIGNAL_4 0x0010
#define MC_PWM_SIGNAL_5 0x0020
#define MC_PWM_NO_SIGNALS 0x0000 /* No (none) PWM signals */
#define MC_PWM_ALL_SIGNALS (MC_PWM_SIGNAL_0 | \
MC_PWM_SIGNAL_1 | \
MC_PWM_SIGNAL_2 | \
MC_PWM_SIGNAL_3 | \
MC_PWM_SIGNAL_4 | \
MC_PWM_SIGNAL_5)
```

Typical Usage of Component in User Code

This chapter describes usage of methods and events that are defined in most hardware oriented components. Usage of other component specific methods is described in the component documentation, in the section "Typical Usage" (*if available*).

Peripheral Initialization Components

Peripheral Initialization Components are the components at the lowest level of peripheral abstraction. These components contain only one method Init providing the initialization of the used peripheral. Refer to the Typical Usage of Peripheral Initialization Components topic for details.

Peripheral Initialization Components

For typical usage and hints on Logical Device Drivers (LDD components), refer to the Typical LDD Components Usage topic.

High Level Components

Methods Enable, Disable

Most of the hardware components support the methods Enable and Disable. These methods enable or disable peripheral functionality, which causes disabling of functionality of the component as well.

TIP Disabling of the peripheral functionality may save processor resources.

Overview of the method behavior according to the component type:

- **Timer components:** timer counter is stopped if it is not shared with another component. If the timer is shared, the interrupt may be disabled (if it is not also shared).
- **Communication components**, such as serial or CAN communication: peripheral is disabled.
- **Conversion components**, such as A/D and D/A: converter is disabled. The conversion is restarted by Enable.

If the component is disabled, some methods may not be used. Refer to components documentation for details.

```
MAIN.C

void main(void)

{

...

B1_Enable(); /* enable the component functionality */

/* handle the component data or settings */

B1_Disable(); /* disable the component functionality */
...
}
```

Methods EnableEvent, DisableEvent

These methods enable or disable invocation of all component events. These methods are usually supported only if the component services any interrupt vector. The method DisableEvent may cause disabling of the interrupt, if it is not required by the component functionality or shared with another component. The method usually does not disable either peripheral or the component functionality.

```
MAIN.C
void main(void)
{
...
B1_EnableEvent(); /* enable the component events */
/* component events may be invoked */
B1_DisableEvent(); /* disable the component events */
/* component events are disabled */
...
}
```

Events BeforeNewSpeed, AfterNewSpeed

Timed components that depend on the microcontroller clock such as timers, communication and conversion components, may support speed modes defined in the processor component (in EXPERT view level). The event BeforeNewSpeed is invoked before the speed mode changes and AfterNewSpeed is invoked after the speed mode changes. Speed mode may be changed using the processor component methods SetHigh, SetLow, or SetSlow.

```
EVENT.C
int changing_speed_mode = 0;
void B1_BeforeNewSpeed(void)
{
++changing_speed_mode;
}
void B1_AfterNewSpeed(void)
{
--changing_speed_mode;
}
```

NOTE If the speed mode is not supported by the component, the component functionality is disabled, as if the method Disable is used. If the supported speed mode is selected again, the component status is restored.

TRUE and FALSE Values of Bool Type

Processor Expert defines the TRUE symbol as 1, however true and false logical values in C language are defined according to ANSI-C:

- False is defined as 0 (zero)
- True is any non-zero value

It follows from this definition, that the bool value cannot be tested using the expressions, such as if (value

== TRUE) ...

Processor Expert methods returning bool value often benefit from this definition and they return non-zero value as TRUE value instead of 1. The correct C expression for such test is: **if (value) ....**

In our documentation, the "true" or "false" are considered as logical states, not any particular numeric values. The capitalized "TRUE" and "FALSE" are constants defined as FALSE=0 and TRUE=1.

Typical Usage of Peripheral Initialization Components

Init Method

Init method is defined in all Peripheral Initialization Components. Init method contains a complete initialization of the peripheral according to the component's settings.

In the following examples, let's assume a component named "Init1" has been added to the project.

The Init method of the Peripheral Initialization component can be used in two ways:

- The Init method is called by Processor Expert
- The Init method is called by the user in his/her module

Automatic Calling of Init

You can let Processor Expert call the `Init` method automatically by selecting "yes" for the Call Init method in the Initialization group of the Component's properties.

When this option is set, Processor Expert places the call of the Init method into the `PE_low_level_init` function of the `CPU.c` module.

Manual Calling of Init

Add the call of the Init method into the user's code, for example in main module.

Enter the following line into the main module file:

```
Init1_Init();
```

Put the Init method right below the `PE_low_level_init` call.

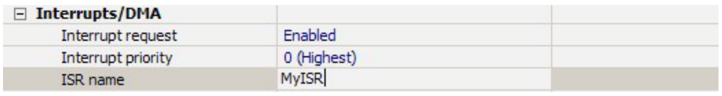
```
void main(void)
{
/*** Processor Expert internal initialization. ***/
PE_low_level_init();
/*** End of Processor Expert internal initialization. ***/
Init1_Init();
for(;;) {}
}
```

Interrupt Handling

Some Peripheral Initialization components allow the initialization of an interrupt service routine. Interrupt(s) can be enabled in the initialization code using appropriate properties that can be usually found within the group *Interrupts*.

After enabling, the specification of an Interrupt Service Routine (ISR) name using the ISR name property is required. This name is generated to Interrupt Vector table during the code generation process. Please note that if the ISR name is filled, it is generated into the Interrupt Vector Table even if the interrupt property is disabled.

Figure 3.29 Example of the Interrupt Configuration



⊟ **Interrupts/DMA**		
Interrupt request	Enabled	
Interrupt priority	0 (Highest)	
ISR name	MyISR	

Enabling/disabling peripheral interrupts during runtime has to be done by user's code, for example by utilizing PESL or direct register access macros, because the Peripheral Initialization Components do not offer any methods for interrupt handling.

The ISR with the specified name has to be declared according to the compiler conventions and fully implemented by the user.

NOTE For 56800/E version users: ISRs generated by Processor Expert contain the fast interrupt handling instructions if the interrupt priority is specified as fast interrupt.

Typical LDD Components Usage

Init method

The `Init()` method is defined in all Logical Device Drivers. The `Init()` method contains a complete initialization of the peripheral according to the component's settings. See Logical Device Drivers for details.

The following example shows how to use Init method in user code, main module in this case. Let's assume a component named "AS1" has been added to the project.

The user needs to add the call of the Init method into the user code, for example in main module.

```
void main(void)

{

LDD_TDeviceStructure MyDevice;

/*** Processor Expert internal initialization. ***/

PE_low_level_init();

/*** End of Processor Expert internal initialization. ***/

MyDevice = AS1_Init(NULL); /* Initialize driver and
peripheral */

. . .

AS1_Deinit(MyDevice); /* Deinitialize driver and peripheral
*/

for(;;) {}

}
```

Deinit Method

`Deinit()` method disables a peripheral and frees the allocated memory if supported by the RTOS adapter. `Deinit()` method is usually used in RTOS applications, not in bare-metal applications.

Interrupt Handling

Most of LDD components are designed to be used in the interrupt mode. It means that the interrupt service routine (ISR) is called by the interrupt controller when an asynchronous interrupt occurs. Interrupt service routine is defined in LDD driver and a user is notified through component’s events. Events can be enabled or disable in the component inspector according to an application needs. When an event is enabled, the appropriate function is generated into Event.c, where a user can write own event handler code. Events are called from the ISR context, so a user should keep an event code as short as possible to minimize a system latency.

User Changes in Generated Code

It's necessary to say at the beginning of the chapter, that modification of the generated code may be done only at user's own risk. Generated code was thoroughly tested by the skilled developers and the functionality of the modified code cannot be guaranteed. We strongly don't recommend modification of the generated code to the beginners. See more information for generated modules in chapter Code Generation.

To support user changes in the component modules, Processor Expert supports the following features:

- Code Generation Options for Component Modules
- Freezing the Code generation

Code Generation Options for Component Modules

It's possible to select mode of the code generation for each component, the following options can be found in the components's pop-up menu in the **Components** view:

- **Always Write Generated Component Modules** (default) - generated component modules are always written to disk and any existing previous module is overwritten
- **Don't Write Generated Component Modules** - the code from component is not generated. Any initialization code of the component, which resides in the processor component, interrupt vector table and shared modules are updated.

Freezing the Code generation

Sometimes, there is unwanted any change in the code generated by Processor Expert. It's for example in case of manual modification done by the user and the user doesn't want to loose the code by accidental re-generation of PE project. For such cases there is an option in Processor Expert Project Options that completely disables the code generation. See Processor Expert Options for details.

Embedded Component Optimizations

This chapter describes how the size and speed of the code could be optimized by choosing right component for the specific task. It also describes how to setup components to produce optimized code. The optimizations that are described are only for the High or Low level components, not for the **Peripheral Initialization components**.

Please refer to sub-chapters for more details:

- General Optimizations
- General Port I/O Optimizations
- Timer Components Optimizations
- Code Size Optimization of Communication Components

General Optimizations

This chapter describes how to setup Processor Expert and components to generate optimized code. The following optimization are only for the High or Low-level components, and not for the Peripheral Initialization components.

Disabling Unused Methods

NOTE These optimization are not usable for the **Peripheral Initialization Components**.

When Processor Expert generates the code certain methods and events are enabled by default setting, even when the methods or events are not needed in the application, and thus while they are unused, the code may still take memory. Basically, the unused methods code is dead stripped by the linker but when the dependency among methods is complex some code should not be dead stripped. When useless methods or events are enabled the generated code can contain spare source code because of these unused methods or events. Moreover some methods can be replaced by more efficient methods that are for special purposes and therefore these methods are not enabled by default.

Disabling Unused Components

Disable unused and test purpose components or remove them from the project. Disabling of these components is sufficient because the useless code is removed but the component setting remains in the project. If these components are required for later testing then add a new configuration to the project and disable these useless component only in the new configuration. The previous configuration will be used when the application is tested again. Moreover if it is required to use the same component with different setting in several configurations, its possible to add one component for each configuration with same name and different setting.

Speed Modes

NOTE These optimizations are not usable for the **Peripheral Initialization Components**.

Timed components which depend on the processor clock (such as timer, communication and conversion components), may support speed modes defined in the processor component (in EXPERT view level). The Processor Expert allows the user to set closest values for the component timing in all speed modes (if possible) . If the requested timing is not supported by the component, for example if the processor clock is too low for the correct function of the component, the component can be disabled for the appropriate speed mode. The mode can be switched in the runtime by a processor method. The component timing is then automatically configured for the appropriate speed mode or the component is disabled (according to the setting). Note, however, that use of speed modes adds extra code to the application. This code must be included to support different clock rates. See speed mode details here.

See chapter Embedded Component Optimizations for details on choosing and setting the components to achieve optimized code.

General Port I/O Optimizations

NOTE These optimizations are not usable for the **Peripheral Initialization Components**.

ByteIO Component Versus BitsIO Component

ByteIO component instead of BitsIO component should be used when whole port is accessed. The BitsIO component is intended for accessing only part of the port (e.g. 4 bits of 8- bit port)

Using the BitsIO component results more complex code because this component provides more general code for the methods, which allows access to only some of the bits of the port. On the other side, the ByteIO component provides access only to the whole port and thus the resulted code is optimized for such type of access.

BitsIO Component Versus BitIO Components

In case of using only a part of the port the multiple BitIO components could be used. A better solution is to use the BitsIO component replacing multiple calls of the BitIO component's methods. The application code consist only of one method call and is smaller and faster.

Timer Components Optimizations

NOTE These optimizations are not usable for the **Peripheral Initialization Components**.

For better **code size performance**, it's recommended to not to use a bigger counter/reload/compare register for timer than is necessary. Otherwise the code size generated by a component may be increased (e.g. For 8-bit timer choose 8bit timer register).

In some cases, several timing periods are required when using timers (For example, the TimerInt component). The Processor Expert allows changing the timer period during run-time using several ways (note that this is an advanced option and the Component Inspector Items visibility must be set to at least 'ADVANCED').

These ways of changing the run-time period of timer requires various amount of code and thus the total application code size is influenced by the method chosen. **When the period must be changed during run-time**, use fixed values for period instead of an interval if possible to save code. There are two possibilities (See Dialog Box for Timing Settings for details.):

- **From list of values** - this allow to specify several (but fixed in run-time) number for given periods. This allows only exact values - modes, listed in the listbox. The resulted code for changing the period is less complex than using an interval.
- **From time interval** - this is an alternative to using 'list of values', which requires more code. Using an interval allows setting whatever value specified by the component during run-time. This code re-calculates the time period to the processor ticks and this value is used when changing the timer period.

If the application requires only a few different timing periods, even if the functionality is the same for both cases, the correct usage of list of periods produces smaller code compared to code using an interval.

Code Size Optimization of Communication Components

NOTE These optimizations are not usable for the **Peripheral Initialization Components**.

Communication components should be used with the smallest possible buffer. Thus the user should compute or check the maximum size of the buffer during execution of the application. For this purpose the method GetCharsInTxBuffer/GetCharsInTxBuffer (AsynchroSerial component), which gets current size of a used buffer, can be used after each time the SendBlock/RecvBlock method is called.

Use interrupts if you require faster application response. The interrupt routine is performed only at the event time, that is the code does not check if a character is sent or received. Thus the saved processor time can be used by another process and application is faster.

Use polling mode instead of interrupts if you require less code because usually overhead of interrupts is bigger than overhead of methods in polling mode. But the polling mode is not suitable for all cases. For example when you use the SCI communication for sending only the data, and a character is sent once in a while, then it is better to use the polling mode instead of using interrupt because it saves the code size, that is when the interrupt is used an interrupt subroutine is needed and code size is increased.

Examples

A module of an application sends once in a while one character to another device through the SCI channel. If the delay between two characters is sufficient to sent one character at a time then the polling mode of the SCI (the AsynchroSerial component) should be used in this case.

A module of an application communicates with another device, that is it sends several characters at one time and receives characters from the device. Thus the interrupt mode of the SCI (the AsynchroSerial component) should be used in this case because when a character is received the interrupt is invoked and the underlying process of the application need not check if a character is received. When a buffer for sending is used, the characters are saved into the buffer and AsynchroSerial's service routine of the interrupt sends these characters without additional code of the application.

NOTE The polling mode of the component is switched on by disabling of the Interrupt service of the component (`AsynchroSerial`, `AsynchroMaster`, and `AsynchroSlave`).

Converting Project to Use Processor Expert

The C project that doesn't use Processor Expert can be can be converted to Processor Expert. This is useful when the user finds out that he/she would like to use additional features of Processor Expert.

WARNING! Note that in most cases this conversion involves necessary manual changes in the application code, because for example the register interrupt vectors table definitions created by the user often conflicts with Processor Expert definitions. Don't forget to backup the whole project before the conversion. Some files will have to be removed from the project. The conversion to Processor Expert is recommended to experienced users only.

The conversion steps are as follows:

1. Select the menu command **File > New > Other..**.
2. Within the "Select a wizard" dialog box select **Processor Expert/Enable Processor Expert for Existing C Project** and click on the **Next** button.
3. Select the project that you would like to convert and the project type.
   – Processor Expert can generate initialization code and drivers for on-chip peripherals and also drivers for selected external peripherals or software algorithms. See Features of Processor Expert for details.
   – Device Initialization is simpler tool that can generate initialization code for on-chip peripherals, interrupt vector table and template for interrupt vector service routines.
4. Select the microcontroller that the project is designed for.
5. Select the microcontroller variant(s) and Processor Expert configuarions that you would like to have available in the project.
6. Review the actions that Processor Expert is about to perform. You can uncheck the checkboxes for items you would like not to be done. Please ensure you have backed-up your project before confirming before you confirm by clicking on Finish.
7. Now it's necessary to move the application code from original `main.c` located in "Sources" folder into new ProcessorExpert.c generated by Processor Expert in previous step, consequently remove original main.c module from the project.
8. For Kinetis family projects, it's necessary to remove the files `kinetis_sysinit.c` and `kinetis_sysinit.h` from `Project_Settings/Startup_Code`. This module contains definitions that conflict with Processor Expert definitions.

Low-level Access to Peripherals

In some cases, a non-standard use of the peripheral is required and it is more efficient to write a custom peripheral driver from scratch than to use the component. In addition, there are special features present only on a particular chip derivative (not supported by the component) that could make the user routines more effective; however, the portability of such code is reduced.

Peripheral Initialization

It is possible to use Processor Expert to generate only the initialization code (function) for a peripheral using the Peripheral initialization components. You can choose a suitable Peripheral initialization component for the given peripheral using the Peripherals tab of the Components Library. Refer to the Components Library View topic for details. Initial values that will be set to the peripheral control registers can be viewed in the Peripheral Initialization window. Refer to the Configuration Registers View topic for details.

Peripheral Driver Implementation

The rest of the peripheral driver can be implemented by the user using one of the following approaches:

- Physical Device Drivers
- Processor Expert System Library
- Direct Access to Peripheral Registers

WARNING! Incorrect use of PESL or change in registers of the peripheral, which is controlled by any Component driver can cause the incorrect Component driver function.

Physical Device Drivers

NOTE PDD layer is available only for Kinetis, MC56F82xxx, and ColdFire+ family microcontrollers.

Physical Device Drivers (PDD) is a software layer that provides set of methods for accessing microcontroller peripheral configuration registers.

PDD methods abstract from: What kind of registers are available

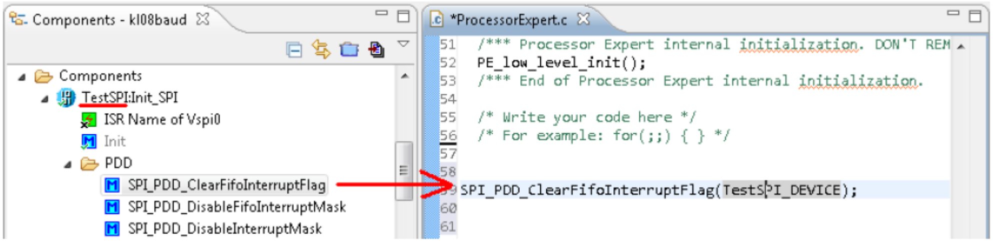
- How registers are organized
- How they are named

PDD header files are located in {InstallDir}\Processor Expert\lib\{MCU}\pdd. Each file contains a definitions of PDD methods for one microcontroller peripheral. Basic PDD methods are implemented by macros and do not provide any additional functionality like register masking, shifting, etc.

Peripheral Base Address

PDD method parameters start with the base address of the peripheral. If user is writing a code which is using a peripheral initialized by the existing component, it is recommend to use the value `{component_name}_DEVICE` which gets replaced by the appropriate address of the selected peripheral for the component. This is automatically done when you drag and drop the PDD method from the PDD sub-folder of the Peripheral Initialization component.

Figure 3.30 Peripheral Base Address


Components - kl08baud
Components
TestSPI:Init_SPI
ISR Name of Vspi0
Init
PDD
SPI_PDD_ClearFifoInterruptFlag
SPI_PDD_DisableFifoInterruptMask
SPI_PDD_DisableInterruptMask
*ProcessorExpert.c

```
/*** Processor Expert internal initialization. DON'T REM
PE_low_level_init();
/*** End of Processor Expert internal initialization.

/* Write your code here */
/* For example: for(;;) { } */


SPI_PDD_ClearFifoInterruptFlag(TestSPI_DEVICE);
```

Otherwise, the peripheral base addresses can be found in the `IO_Map.h` file.

Processor Expert System Library

NOTE PESL is supported only for 56800/E

PESL (Processor Expert System Library) is dedicated to power programmers, who are familiar with the microcontroller architecture - each bit and each register. PESL provides macros to access the peripherals directly. It should be used only in special cases when the low-level approach is necessary.

PESL is peripheral oriented and complements with Embedded Components, which are functionality oriented. While Embedded Components provide very high level of project portability by stable API and inheritance feature across different CPU/DSP/PPC architectures, PESL is more architecture dependent.

PESL commands grouped by the related peripheral can be found in Processor Expert **Components** view in PESL folder.

Convention for PESL Macros

Each name of the PESL macro consists of the following parts:

```
PESL(device name, command, parameter)
```

Example:

```
PESL(SCI0, SCI_SET_BAUDRATE, 0);
```

Using PESL and Peripheral Initialization Components

For every Peripheral Initialization Component (for details, refer to the Component Categories topic) there is a C macro defined by Processor Expert with the name ***component name*****`_DEVICE`**. This macro results to the name of the peripheral selected in the component named 'component name'. Using this macro instead of a real peripheral name allows a peripheral to be changed later by changing the component property without modifying the PESL commands in user code.

Example:

Let's expect we have a component Init_SCI named SCI1:

```
PESL(SCI1_DEVICE, SCI_SET_BAUDRATE, 1);
```

Processor Expert shows the list of the available PESL commands as a subtree of the Peripheral Initialization component in the Components view (refer to the Processor Expert Options topic for details). User can drag and drop the commands into the code from this tree. The PESL commands created this way use the component name `_DEVICE` macro instead of a specific peripheral name.

PESL Commands Reference

For details on PESL, its commands and parameters, see PESL Library user manual using the **Help** command of PESL folder pop-up menu.

Direct Access to Peripheral Registers

NOTE Register access macros are not available for Kinetis and ColdFire+ family microcontrollers.

The direct control of the Peripheral's registers is a low-level way of creating peripheral driver which requires a good knowledge of the target platform and the code is typically not portable to different platform. However, in some cases is this method more effective or even necessary to use (in the case of special chip features not encapsulated within the

Embedded component implementation). Refer to the Low-level Access to Peripherals topic for details.

The common basic peripheral operations are encapsulated by the PESL library commands which is effectively implemented using the simple control register writes. Refer to the Processor Expert System Library topic for details.

Register Access Macros

Processor Expert defines a set of C macros providing an effective access to a specified register or its part. The definitions of all these macros are in the file **`PE_Types.h`**. The declaration of the registers which could be read/written by the macros is present in the file **`IO_Map.h`**.

Whole Register Access Macros

- **getReg{w}** (*RegName*) — Reads the register content
- **setReg{w}** (*RegName*, *RegValue*) — Sets the register content

Register Part Access Macros

- **testReg{w}Bits** (*RegName*, *GetMask*) — Tests the masked bits for non-zero value
- **clrReg{w}Bits** (*RegName*, *ClrMask*) — Sets a specified bits to 0.
- **setReg{w}Bits** (*RegName*, *SetMask*) — Sets a specified bits to 1.
- **invertReg{w}Bits** (*RegName*, *InvMask*) — Inverts a specified bits.
- **clrSetReg{w}Bits** (*RegName*, *ClrMask*, *SetMask*) — Clears bits specified by ClrMask and sets bits specified by SetMask

Access To Named Bits

- **testReg{w}Bit** (*RegName*, *BitName*) — Tests whether the bit is set.
- **setReg{w}Bit** (*RegName*, *BitName*) — Sets the bit to 1.
- **clrReg{w}Bit** (*RegName*, *BitName*) — Sets the bit to 0.
- **invertReg{w}Bit** (*RegName*, *BitName*) — Inverts the bit.

Access To Named Groups of Bits

- **testReg{w}BitGroup** (*RegName*, *GroupName*) — Tests a group of the bit for non-zero value

- **getReg{w}BitGroupVal** (*RegName*, *GroupName*) — Reads a value of the bits in group
- **setReg{w}BitGroupVal** (*RegName*, *GroupName*, *GroupVal*) — Sets the group of the bits to the specified value.

RegName - Register name

BitName - Name of the bit

GroupName - Name of the group

BitMask - Mask of the bit

BitsMask - Mask specifying one or more bits

BitsVal - Value of the bits masked by BitsMask

GroupMask - Mask of the group of bits

GetMask - Mask for reading bit(s)

ClrMask - Mask for clearing bit(s)

SetMask - Mask for setting bit(s)

InvMask - Mask for inverting bit(s)

RegValue - Value of the whole register

BitValue - Value of the bit (0 for 0, anything else = 1)

{w} - Width of the register (8, 16, 32). The available width of the registers depends on used platform.

Example

Assume that you have a processor which has a PWMA channel and it is required to set three bits (0,1,5) in the PWMA_PMCTL to 1. Use the following line:

```
setRegBits(PWMA_PMCTL,35); /* Run counter */
```

Processor Expert Files and Directories

PE Project File

All components in the project with their state and settings and all configurations are stored in one file **`ProcessorExpert.pe`** in the root of project directory. If the whole content of the project including subdirectories is copied or moved to another directory, it is still possible to open and use it in the new location.

Project Directory Structure

Processor Expert uses the following sub-directory structure within the project directory:

- **\Generated_Code** — the directory containing all generated source code modules for components.
- **\Documentation** — the directory with the project documentation files generated by Processor Expert.
- **\Sources** — the directory for main module, event module other user modules.
- **\Static_Code** — the directory containing all static source code modules. For details, refer to the Static Code Support in Processor Expert topic.

For details on files generated by Processor Expert, refer to the Code Generation topic.

User Templates and Components

User-created templates (refer to the Creating User Component Templates topic) and components are shared by all users and they are stored in the directory:

`%ALLUSERSPROFILE%\ApplicationData\Processor Expert\{version}\`

For example `C:\Documents and Settings\All Users\ApplicationData\Processor Expert\CW08_PE3_02\`

Static Code Support in Processor Expert

This topic describes static initialization support in Processor Expert. For selected microcontrollers, Processor Expert offers static (that is not generated), initialization code consisting of Processor Expert startup code, Peripheral Initialization Components and CPU (processor) component code, Peripheral Device Drivers (PDD) and peripheral memory map. This feature is currently available for MC56F8200, Kinetis V, Kinetis W families and MK24, MK63, MK64, MKE06Z128/64, MKE04Z128/64 and SKEAZ128/64 derivatives. Rest of Kinetis derivatives support only PDD and peripheral memory map static code.

Static Initialization Code

For selected families of processors, peripheral initialization and processor components use static peripheral initialization drivers. Each instance of MCU peripheral has its own initialization driver placed in static (that is not generated) module. Processor module, which covers MCU startup, is also implemented using static code.

Peripheral Initialization Components covers all MCU peripherals, with exception for system and bus clock generator peripherals, which are controlled by processor component as they are part of Processor Expert timing model and their initialization is done during the startup.

Each Peripheral Initialization Component configures one peripheral and covers all its features based on the full set of peripheral registers. Peripheral registers are initialized in the component static initialization method according to component settings represented in configuration header file which is included in static driver. Refer to the Typical Usage of Peripheral Initialization Components section for more details about peripheral initialization component initialization method.

Some additional features associated with the peripheral functionality but controlled by registers belonging to different peripherals might be also included in component settings, for example pin routing, interrupts configuration or clock gate control. Such settings lead to initialization of registers not covered by component itself (refer to the Multiple Properties Accessing Common Register and <Automatic> Value section for more information on accessing register from multiple properties). Initialization of these additional registers is handled by Processor Expert depending on whether Peripheral Initialization Component covering the registers is present in the project or not:

- Project contains Peripheral Initialization Component which covers initialization of peripheral to which the additional registers belong. In this case, the additional registers initialization is included in the static initialization method of this component.
- Project doesn't contain Peripheral Initialization Component which would be covering initialization of additional registers. In such case, Processor Expert takes care of initialization of registers not covered by any initialization component and generates appropriate code to the `Common_Init()` method. User is informed about this by warning generated by Processor Expert.

For example, application uses `Init_ADC` component to configure A/D converter with measurement synchronized with external device connected using pin on chip package. Routing of the pin involves initialization of General Purpose I/O port A (GPIOA) peripheral. If `Init_GPIO` component configuring GPIOA peripheral is in the project then GPIOA registers involved in `Init_ADC` pin routing are initialized in the `Init_GPIO` static initialization method. If `Init_GPIO` for GPIOA is not used GPIOA registers initialization is generated into the `Common_Init()`.

Standalone and Linked Project

Static code support allows choosing one of two modes for each created project:

- Standalone, in which project's static code is independent from currently used development environment.
- Linked, which uses shared static code.

Both modes use shared and project static code repository. Project repository is placed in the project directory (refer to the Project Static Code Directory Structure for more details) and shared repository is placed in the Processor Expert installation directory (refer to the Processor Expert Static Code Repository and Project Repository Synchronization section). In standalone mode, project repository contains copy of shared repository created during project creation. In linked mode, project repository links the shared one. It is not possible to change mode, once it is selected.

Standalone Mode

Complete set of static drivers is stored directly in the project directory. During project creation, static drivers are copied from shared static code repository to the project repository. After project is created, it is independent from development environment and from other projects therefore changes made in static drivers in one project don't affect rest of the projects.

It is possible to synchronize project's local repository with the common one - Processor Expert detects changes in the common repository, informs user about them and offers project local repository update. Also, if static driver is missing in the project directory but is required by Embedded Component used in the project then Processor Expert offers adding of the missing files.

Linked Mode

Static drivers are not physically placed in the project directory. Instead, project static code repository is virtually linked with shared static drivers repository stored in Processor Expert installation directory. All projects created in this mode shares the same version of static drivers and any changes made in the shared repository are automatically distributed across all of the linked projects.

Processor Expert Static Code Repository

Processor Expert installation directory contains static code repository with static driver modules for all supported processors. Projects in linked mode shares the repository and repository content is used when new standalone project is created. Refer to the Standalone and Linked Project section for more information about standalone and linked mode.

Processor Expert static code repository is divided into sub-folders with one processor family support in each of them. It is placed at: `{CodeWarrior}\Processor Expert\lib\{Processor family}` and contains among others following sub-directories:

- `iofiles` — the directory containing peripheral memory map files
- `pdd\inc` — the directory containing Physical Device Drivers (PDD) header files. For more information on PDD, refer to the Physical Device Drivers section.

- `pdd2\{Processor}\peripherals` — the directory containing all peripheral initialization modules.
- `pdd2\{Processor}\system` — the directory with system and common source code modules including processor modules.

Project Static Code Directory Structure

Processor Expert uses the following sub-directory structure within the project directory:

- `Generated_Code` — the directory containing all generated source code modules for components. Among others, this directory stores components interface headers and configuration header files included in the static modules for static code parameterization.
- `Documentation` — the directory with the project documentation files generated by Processor Expert.
- `Sources` — the directory for main module, event module and other user modules.
- `Static_Code` — the directory containing project static code repository with all static source code modules. Project static code repository contains either copy (in case of standalone project) or link (in case of linked project) to Processor Expert static code repository.

Static source code sub-directory contains:

- `IO_Map` — the directory with processor peripheral memory map file.
- `PDD` — the directory containing Physical Device Drivers (PDD) header files. For more information on PDD, refer to the Physical Device Drivers section.
- `Peripherals` — the directory containing all peripheral initialization modules.
- `System` — the directory with system and common source code modules including processor modules.

Project Static Modules

Embedded Components with static code support uses or creates several project files. These files can be divided into following groups:

- Processor component files
- Peripheral Initialization Component files
- Peripheral memory map files
- Common project files

Each Embedded Component with static code implements following modules:

- Static modules with peripheral initialization code - module name has `_Init` suffix.

- Configuration header files with parameterization of the static module - header file name has `_Config` suffix. Parameterization constant are controlled by component properties.
- Generated component interface header file - header file has the same name as the component.

Processor Component Files

Static modules:

- `CPU_Init.c` — the file contains startup initialization code and processor runtime methods code.
- `CPU_Init.h` — the file contains declaration of runtime methods and their input parameter types.

For more information about processor runtime methods see documentation of specific processor component. Detailed startup code description is explained in the Reset Scenario section of Implementation Details.

Processor Expert static code repository: `{CodeWarrior}\Processor Expert\lib\{Processor family}\pdd2\{Processor}\system`

Project static code repository: `{Project}\Static_Code\System`

Generated modules:

- `CPU_Config.h` — configuration header file contains `CPU_Init.c` module parameterization constants.
- `{Processor component name}.c` — the file contains `Common_Init()` and `Components_Init()` methods. Refer to the Reset Scenario section of Implementation Details for more details about these generated methods.
- `{Processor component name}.h` — the file contains declaration of `Common_Init()` and `Components_Init()` along with clock configuration constants.
- Generated modules are placed at: `{Project}\Generated_Code`

Peripheral Initialization Component Files

Static modules:

- `{Peripheral}_Init.c` — the file contains peripheral initialization method.
- `{Peripheral}_Init.h` — the file contains peripheral initialization method declaration.

Processor Expert static code repository: `{CodeWarrior}\Processor Expert\lib\{Processor family}\pdd2\{Processor}\peripherals`

Project static code repository: `{Project}\Static_Code\Peripherals`

Generated modules:

- `{Peripheral}_Config.h` — configuration header file contains `{Peripheral}_Init.c` module parameterization constants. The constants reflect settings of the Embedded Component.
- `{Initialization component name}.h` — application interface header file. Intended to be included in the application code to map user-defined initialization method name with its static implementation. Also contains selected-device-independent component-specific peripheral base address constant used as first parameter of PDD macros. If interrupts are enabled, header also contains component interrupt subroutine (ISR) declaration.

Generated modules are placed at: `{Project}\Generated_Code`

Peripheral Memory Map Files

Static modules:

- `{Processor variant}.h` — the file contains processor peripheral memory map implementation

Processor Expert static code repository: `{CodeWarrior}\Processor Expert\lib\{Processor family}\iofiles`

Project static code repository: `{Project}\Static_Code\IO_Map`

Generated modules:

- `IO_Map.h` — application interface header file. Intended to be included in the application code to access processor-specific peripheral memory map.

Generated modules are placed in the: `{Project}\Generated_Code`

Common Project Files

Static modules:

- `Vectors.c` — the file contains interrupt vector table and unused interrupts subroutines implementation.
- `Peripherals_Init.c` — the file contains `Peripherals_Init()` method which calls all peripheral initialization methods supported for selected processor. Refer to the Reset Scenario section of Implementation Details for details on `Peripherals_Init()` method.
- `Peripherals_Init.h` — the file contains `Peripherals_Init()` method declaration.

- `PDD_Includes.h` — the file includes all PDD modules supported for selected processor. Intended to be included in the application code to access PDD macros. For more information on PDD, refer to the Physical Device Drivers.
- `PE_Types.h` — the file contains common Processor Expert types, constants and macros implementation.
- `PE_Error.h` — the file contains error code constants.

Processor Expert static code repository: `{CodeWarrior}\Processor Expert\lib\{Processor family}\pdd2\{Processor}\system`

Project static code repository: `{Project}\Static_Code\System`

Generated modules:

- `Vectors_Config.h` — configuration header file contains `Vectors.c` module parameterization constants.
- `Init_Config.h` — the file includes all components configuration and interface header files.

Generated modules are placed in the: `{Project}\Generated_Code`

Processor Expert Static Code Repository and Project Repository Synchronization

During project creation In the Standalone project mode, the static source modules are copied from Processor Expert (common) static code repository to the project repository placed in `Static_Code` project directory. This secures project code independence. However, Processor Expert detects changes between common and project repository and allows user to update project static code source files from common repository.

Differences between project and common repository are checked during code generation and user is informed about them in the **Track Changes** dialog window. In this window, user can view list of all files from common repository that has been modified, check changes in their code and choose whether the modification of project file should be accepted or not.

NOTE IO_Map and System sub-folders are always checked for differences. PDD and Peripherals sub-folders are checked only if associated Peripheral Initialization component is in the project. Refer to the Project Static Code Directory Structure for more information on static code sub-folders.

Typically, static code changes when processor component or compiler is switched.

It is also possible to force project and common repository synchronization explicitly using Synchronize Processor Expert Static Code command available in Project menu or

Components View menu. In this case, all static repository files are showed in the Track Changes window with their status. Repository synchronization can be used for example to update project after Processor Expert update or when project should use different Processor Expert version.

Static Code Track Changes

Static Code Track Changes window shows any differences between files in the common and project repository. Window contains four columns:

- **Source File Name** — Common repository file name including full path.
- **Action** — Action to be performed.
- **Project File Name** — Project repository file name including path in the project directory.
- **Status** — Description of detected modification.

Static file status can be:

- **Source file changed** — Changes detected in the common repository file.
- **No more needed** — File has been removed from common repository.
- **Modified by the user** — Project repository file has been modified.
- **Project file not found** — New file in common repository was found or project repository file has been removed.
- **Up-to-date** — No difference between source and project file detected. Showed only during synchronization forced by Synchronize Processor Expert Static Code command.

Available actions:

- **Update/Do not update** — Update common repository file or leave it unchanged.
- **Create/Do not create** — Copy file from common to project repository or don't copy it.
- **Delete/Do not delete** — Remove file from project repository or keep it.
- **No action** — No action needed. Showed only during synchronization forced by Synchronize Processor Expert Static Code command in case that there is no difference between source and project file (Up-to-date status).

Compare file by content:

If difference between common and project repository file is detected then changes in the file can be showed using **Compare file by content** button. This will open Eclipse base C Compare Viewer panel inside the Track Changes window and highlighting each change. This way detailed review of changes can be done.

Multiple Properties Accessing Common Register and `<Automatic>` Value

Embedded Components are functional-oriented therefore allows settings that do not affect only one peripheral but several of them. However, it is also possible to access each peripheral individually, one-by-one using the Peripheral Initialization Components. This means that some specific hardware functionalities can be accessed from multiple components. During application design, Processor Expert verifies that component properties accessing the same functionality are not in conflict with other properties and if conflict is detected then informs the user.

For example, pin routing involves configuration of multiple registers across different peripherals and it is possible to set it manually for each involved peripheral or just set required pin in pin property of Embedded Component that needs the pin for its function (it consumes the pin). Processor Expert searches its knowledge base for initialization value of all registers involved in pin routing and searches the project for any Embedded Component which would also initialize such registers. If any component is found then it is checked that its settings doesn't produce initialization which would be in conflict with initialization required to route pin used by the consuming component. If any conflict in initialization is detected user is informed by error showed for each property which affects the routing and user has to resolve them. This is when `<Automatic>` property value can be used.

Properties which affects common hardware functionality allows setting `<Automatic>` value instead of using specific setting. `<Automatic>` value signals that property doesn't require specific setting and leaves the control of hardware functionality to different property to specify it. Therefore, property set to `<Automatic>` is not causing any conflicts. All properties that allow `<Automatic>` value have it as a default state after component is added to the project.

In example, described above means that if consumer property defines required pin and at the same time properties involved in its routing have `<Automatic>` set then there are no conflicts or errors.

When `<Automatic>` is set in the property then currently used value is controlled by different property, however the value is showed next to the automatically controlled property in the **Details** column in the **Component Inspector** view. Also, if the property control registers which are initialized by the component's initialization method then the currently used value is propagated through the property into the initialization method.

Internal signals

Internal signals represent interconnections between processor internal peripherals, a peripheral produces signal which is used by different peripherals. Examples of internal signals are:

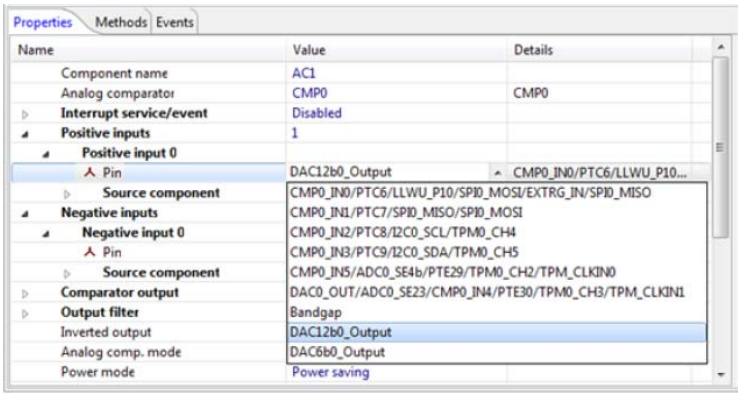
- Trigger — signal asserting request (example of trigger source is Program Delay Block or any timer).
- Internal voltage sources — internal reference or programmable voltage sources (for example, produced by D/A converter).

In Processor Expert, each internal signal is represented by name, which is usually composed of name of peripheral producing the signal and function of the signal. Some examples are `CMP0_output`, `TMR0_overflow`, and `DAC6b_output`. However, it is not guaranteed that the same internal signal will have exactly the same representation across all components. In embedded components, internal signals represented by their name are typically present on the consumer peripheral side as inputs to the embedded components. On the signal producer side, internal signal generation control is peripheral specific.

Some examples of internal signals available on KL25 Kinetis MCU:

Analog comparator input — (CMP) peripheral can compare two types of analog signal sources: external, connected through package pin, or internal, connected using internal signal. Figure below shows list of sources available for the `CMP0` in the `AnalogComp_LDD` component.

Figure 3.31 Analog comparator input selection in AnalogComp_LDD



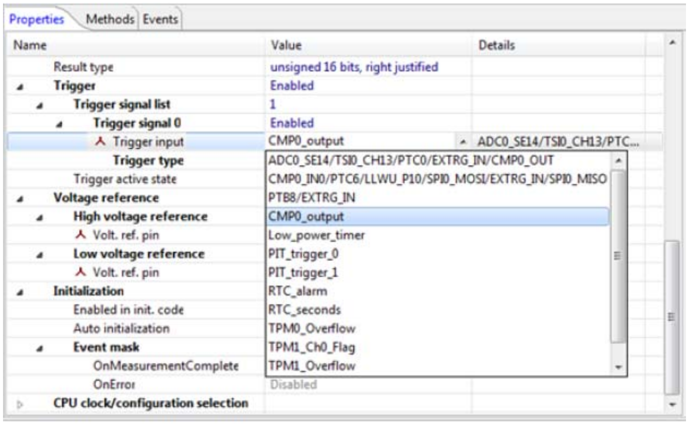
Name	Value	Details
Component name	AC1	
Analog comparator	CMP0	CMP0
Interrupt service/event	Disabled	
Positive inputs	1	
Positive input 0		
Pin	DAC12b0_Output	CMP0_IN0/PTC6/LLWU_P10...
Source component		
Negative inputs		
Negative input 0		
Pin		
Source component		
Comparator output		
Output filter		
Inverted output		
Analog comp. mode		
Power mode	Power saving	

CMP0_IN0/PTC6/LLWU_P10/SPI0_MOSI/EXTRG_IN/SPI0_MISO
CMP0_IN1/PTC7/SPI0_MISO/SPI0_MOSI
CMP0_IN2/PTC8/I2C0_SCL/TPM0_CH4
CMP0_IN3/PTC9/I2C0_SDA/TPM0_CH5
CMP0_IN5/ADC0_SE4b/PTE29/TPM0_CH2/TPM_CLKIN0
DAC0_OUT/ADC0_SE23/CMP0_IN4/PTE30/TPM0_CH3/TPM_CLKIN1
Bandgap
DAC12b0_Output
DAC6b0_Output

From the list, internal signals are `Bandgap` (constant voltage reference), `DAC12b0_Output` (output of the 12-bit DA converter 0), and `DAC6b0_Output` (output of the 6-bit DA converter 0).

Analog-to-digital converter (ADC) trigger — A/D conversion can be triggered from the package pin or from number of internal peripherals. Figure below shows some of the triggers offered in the `ADC_LDD` component **Trigger input** property.

Figure 3.32 A/D converter trigger selection in ADC_LDD



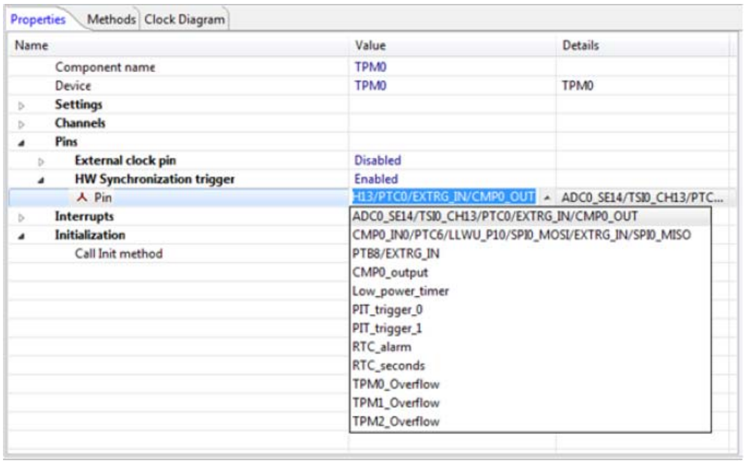
Name	Value	Details
Result type	unsigned 16 bits, right justified	
Trigger	Enabled	
Trigger signal list	1	
Trigger signal 0	Enabled	
Trigger input	CMP0_output	ADC0_SE14/TSI0_CH13/PTC...
Trigger type		
Trigger active state		
Voltage reference		
High voltage reference		
Volt. ref. pin		
Low voltage reference		
Volt. ref. pin		
Initialization		
Enabled in init. code		
Auto initialization		
Event mask		
OnMeasurementComplete		
OnError	Disabled	
CPU clock/configuration selection		

```
ADC0_SE14/TSI0_CH13/PTC0/EXTRG_IN/CMP0_OUT
CMP0_IN0/PTC6/LLWU_P10/SPI0_MOSI/EXTRG_IN/SPI0_MISO
PTB8/EXTRG_IN
CMP0_output
Low_power_timer
PIT_trigger_0
PIT_trigger_1
RTC_alarm
RTC_seconds
TPM0_Overflow
TPM1_Ch0_Flag
TPM1_Overflow
```

From the list, internal signals are `CMP0_output` (Comparator 0 output), `Low_power_timer` (Low power timer event), `PIT_trigger_0/1` (Periodic interrupt timer 0 and 1 event), `RTC_alarm` (Real-time clock alarm event), `RTC_seconds` (Real-time clock second event), `TPM0_Overflow` (Timer PWM module 0 overflow event), `TPM1_Ch0_Flag` (Timer PWM module 0 channel 0 event) and `TPM1_Overflow` (Timer PWM module 1 overflow event).

Timer synchronization — some timers can be synchronized by trigger. Figure below shows how selection of appropriate trigger looks like in the `Init_TPM` component. This example also shows that selection of internal signals in Peripheral Initialization components is same as in high-level and LDD components.

Figure 3.33 Timer synchronization trigger selection in Init_TPM


Properties Methods Clock Diagram

Name	Value	Details
Component name	TPM0	
Device	TPM0	TPM0
Settings		
Channels		
Pins		
External clock pin	Disabled	
HW Synchronization trigger	Enabled	
Pin	H13/PTC0/EXTRG_IN/CMP0_OUT	ADC0_SE14/TSI0_CH13/PTC...
Interrupts		
Initialization		
Call Init method		

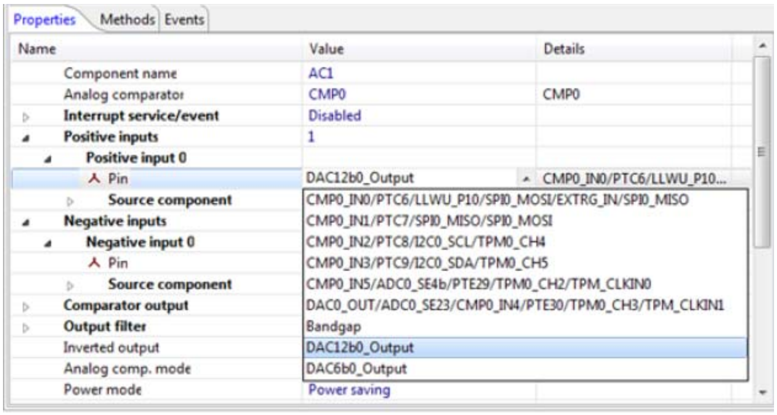
ADC0_SE14/TSI0_CH13/PTC0/EXTRG_IN/CMP0_OUT
CMP0_IN0/PTC6/LLWU_P10/SPI0_MOSI/EXTRG_IN/SPI0_MISO
PTB8/EXTRG_IN
CMP0_output
Low_power_timer
PIT_trigger_0
PIT_trigger_1
RTC_alarm
RTC_seconds
TPM0_Overflow
TPM1_Overflow
TPM2_Overflow

From the list, internal signals are `CMP0_output` (Comparator 0 output), `Low_power_timer` (Low power timer event), `PIT_trigger_0/1` (Periodic interrupt timer 0 and 1 event), `RTC_alarm` (Real-time clock alarm event), `RTC_seconds` (Real-time clock second event), `TPM0/1/2_Overflow` (Timer PWM module 0, 1 and 2 overflow event).

Differentiation of Internal Signals and Pins

It is often possible to choose peripheral input between internal signals and package pins. For example, figure below shows comparator input source selection offered in the `AnalogComp_LDD` component. There is list of pins and internal signals that can be used as comparator input. In Processor Expert, only pin names uses / (slash) separator (to distinguish individual pin functions). Internal signals do not contain this symbol in their name. This is how pins can be easily identified.

Figure 3.34 Multiple analog comparator inputs in AnalogComp_LDD

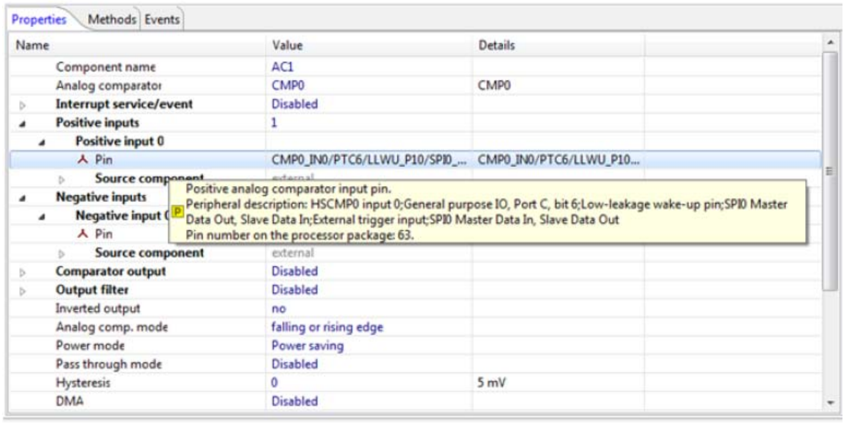

Properties | Methods | Events

Name	Value	Details
Component name	AC1	
Analog comparator	CMP0	CMP0
Interrupt service/event	Disabled	
Positive inputs	1	
Positive input 0		
Pin	DAC12b0_Output	CMP0_IN0/PTC6/LLWU_P10...
Source component		
Negative inputs		
Negative input 0		
Pin		
Source component		
Comparator output		
Output filter		
Inverted output		
Analog comp. mode		
Power mode	Power saving	

CMP0_IN0/PTC6/LLWU_P10/SPI0_MOSI/EXTRG_IN/SPI0_MISO
CMP0_IN1/PTC7/SPI0_MISO/SPI0_MOSI
CMP0_IN2/PTC8/I2C0_SCL/TPM0_CH4
CMP0_IN3/PTC9/I2C0_SDA/TPM0_CH5
CMP0_IN5/ADC0_SE4b/PTE29/TPM0_CH2/TPM_CLKIN0
DAC0_OUT/ADC0_SE23/CMP0_IN4/PTE30/TPM0_CH3/TPM_CLKIN1
Bandgap
DAC12b0_Output
DAC6b0_Output

However, Processor Expert offers clear differentiation between internal signals and pins, as shown in figures below.

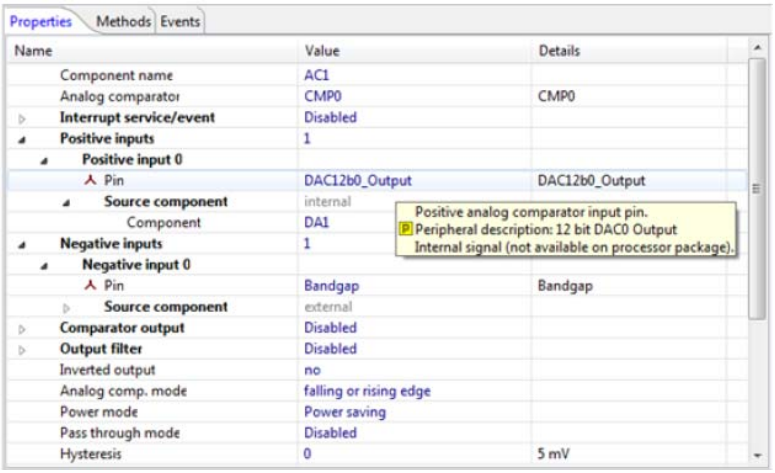
Figure 3.35 Pin selection


Properties | Methods | Events

Name	Value	Details
Component name	AC1	
Analog comparator	CMP0	CMP0
Interrupt service/event	Disabled	
Positive inputs	1	
Positive input 0		
Pin	CMP0_IN0/PTC6/LLWU_P10/SPI0_...	CMP0_IN0/PTC6/LLWU_P10...
Source component	external	
Negative inputs		
Negative input 0		
Pin		
Source component	external	
Comparator output	Disabled	
Output filter	Disabled	
Inverted output	no	
Analog comp. mode	falling or rising edge	
Power mode	Power saving	
Pass through mode	Disabled	
Hysteresis	0	5 mV
DMA	Disabled	

Positive analog comparator input pin.
Peripheral description: HSCMP0 input 0;General purpose IO, Port C, bit 6;Low-leakage wake-up pin;SPI0 Master Data Out, Slave Data In;External trigger input;SPI0 Master Data In, Slave Data Out
Pin number on the processor package: 63.

Figure 3.36 Internal signal selection


Properties Methods Events

Name	Value	Details
Component name	AC1	
Analog comparator	CMP0	CMP0
Interrupt service/event	Disabled	
Positive inputs	1	
Positive input 0		
Pin	DAC12b0_Output	DAC12b0_Output
Source component	internal	
Component	DA1	
Negative inputs	1	
Negative input 0		
Pin	Bandgap	Bandgap
Source component	external	
Comparator output	Disabled	
Output filter	Disabled	
Inverted output	no	
Analog comp. mode	falling or rising edge	
Power mode	Power saving	
Pass through mode	Disabled	
Hysteresis	0	5 mV

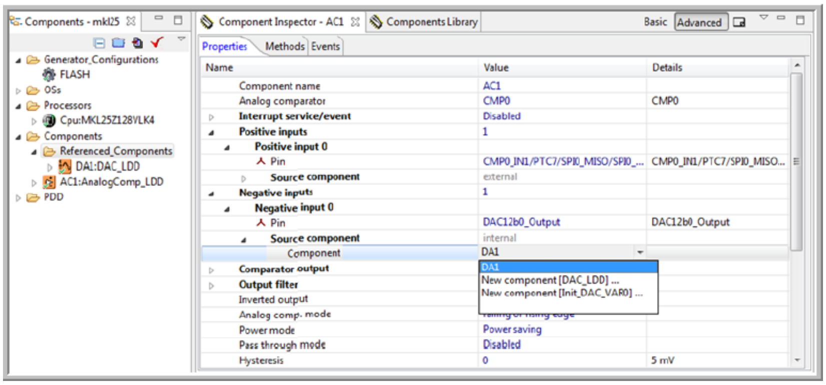
Positive analog comparator input pin.
Peripheral description: 12 bit DAC0 Output
Internal signal (not available on processor package).

Clear differentiation is possible in property hint. In case of pins, full description of pin functions along with pin number on processor package is showed. If internal signal is selected then this fact is signalized at the last line of the hint.

Internal signal producer component linking

When internal signal is used in the high level or LDD component, the component producing the signal might be required by the component. In such case, special component-linking property is used to add required component. This feature is used to guide user to the components that are necessary for correct function of peripheral that depends on signal generated by different peripherals as shown in figure below.

Figure 3.37 Example of component linking in AnalogComp_LDD



Name	Value	Details
Component name	AC1	
Analog comparator	CMP0	CMP0
Interrupt service/event	Disabled	
Positive inputs	1	
Positive input 0		
Pin	CMP0_IN1/PTC7/SPI0_MISO/SPI0_...	CMP0_IN1/PTC7/SPI0_MISO...
Source component	external	
Negative inputs	1	
Negative input 0		
Pin	DAC12b0_Output	DAC12b0_Output
Source component	internal	
Component	DA1	
Comparator output		
Output filter		
Inverted output		
Analog comp. mode		
Power mode	Power saving	
Pass through mode	Disabled	
Hysteresis	0	5 mV

Figure shows `AnalogComp_LDD` component that uses pin as positive input and internal signal from 12-bit D/A converter (DAC) output as negative input. Selection of internal signal causes that Source component property from the negative input group is active and DAC component is required. User may add new DAC component (`DAC_LDD` or `Init_DAC` in this case) or use `DAC_LDD` component already present in the project (component DA1).

4

Processor Expert Tutorials

This tutorial is provided for embedded system designers who wish to learn how to use the features of Processor Expert. This tutorial will help you to start using Processor Expert for your own application.

This chapter explains:

Tutorial Project 1 for Kinetis Microcontrollers

Tutorial Project 1 for Kinetis Microcontrollers

This simple tutorial describes a periodically blinking LED project. The LED is connected to one pin of the processor and it is controlled by a periodical timer interrupt.

The project is designed to work with PK60N512 processor and TWR-K60N512 tower board. However, it is not necessary to have this hardware. The project can be created without it.

This simple Processor Expert demo-project uses the following LDD Embedded Components (refer to the Component Categories topic for details):

1. PK60N512 — processor component
2. GPIO_LDD — This component will control LED output connected to PTA10 pin.
3. TimerUnit_LDD — This component will provide periodical timing.

This tutorial contains the following main steps:

1. Creating a New Project
2. Adding Components
3. Configuring Components
4. Code Generation

Creating a New Project

To create a new project:

1. In the IDE, click **File** menu and select **New > Bareboard Project** in order to create a new project.
2. The Project Wizard appears. Enter the name of the project *LED* and click **Next**.

NOTE You can also create a project outside Eclipse workspace. In the **Create an MCU bareboard project** page, uncheck the **User default location** checkbox and specify the location. This option will allow you to create a project and generate the code that will be compiled by external compiler and is not integrated in Eclipse.

3. Select **Kinetis K Series > K6x Family > K60D/PK60N512**. Click **Next**.
4. In the **Connections** page, **P&E USB BDM Multilink Universal [FX]/USB Multilink** is set as the default connection. Click **Next**.
5. In the **Language and Build Tool Options** page, set **C** in **Languages** page. Click **Next**.
6. In the **Rapid Application Development** page, select the **Processor Expert** from **Rapid Application Development** group and click **Finish**.

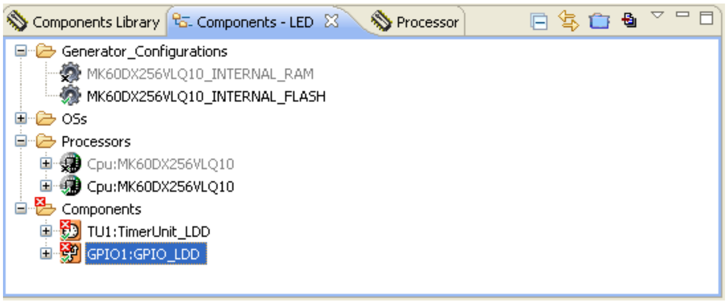
The new project is created and ready for adding new components. For adding new components, refer to the Adding Components topic.

Adding Components

1. In the **Components Library** view, switch to **Alphabetical** tab and find a **GPIO_LDD** component and from its pop-up menu (invoked by the right mouse button) select **Add to project** option.
2. Find **TimerUnit_LDD** component and from its pop-up menu (invoked by the right mouse button) select **Add to project** option.

The components are now visible in the **Components** folder in the **Components** view.

Figure 4.1 Component View



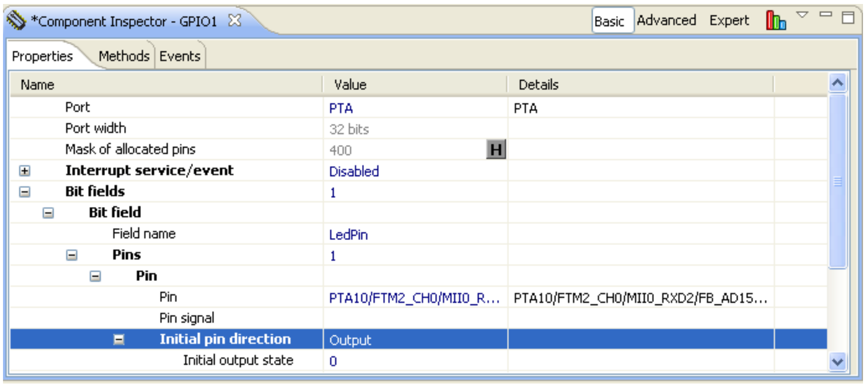
```
Generator_Configurations
    MK60DX256VLQ10_INTERNAL_RAM
    MK60DX256VLQ10_INTERNAL_FLASH
OSs
Processors
    Cpu:MK60DX256VLQ10
    Cpu:MK60DX256VLQ10
Components
    TU1:TimerUnit_LDD
    GPIO1:GPIO_LDD
```

For configuring new components, refer to the Configuring Components topic.

Configuring Components

1. In the **Components** view, click **GPIO1:GPIO_LDD** component to open it in **Component Inspector** view.
2. In the **Component Inspector** view, set the property **Field name** in the first **Bit field** group to **LedPin**. From the drop-down menu, select the value for **Pin** as **PTA10** (it corresponds to LED17 on the tower board). Set the **Initial pin direction** property to **Output**. Set **Auto initialization** to **yes.**

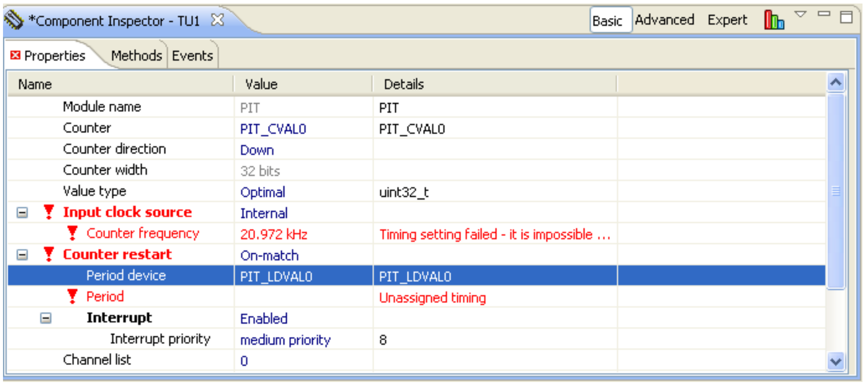
Figure 4.2 Component Inspector



Name	Value	Details
Port	PTA	PTA
Port width	32 bits	
Mask of allocated pins	400	
Interrupt service/event	Disabled	
Bit fields	1	
Bit field		
Field name	LedPin	
Pins	1	
Pin		
Pin	PTA10/FTM2_CH0/MII0_R...	PTA10/FTM2_CH0/MII0_RXD2/FB_AD15...
Pin signal		
Initial pin direction	Output	
Initial output state	0	

3. In the **Components** view, click **TU1:TimerUnit_LDD** component to open it in the **Component Inspector**.
4. Set the following properties:
   - **Counter: PIT_CVAL0**
   - **Counter direction: Down**
   - **Counter restart: On-match** — allows to set desired period of interrupt, otherwise interrupt invocation period is fixed to counter overflow.
   - **Period: 500 ms**
   - **Interrupt: Enabled**
   - **Auto initialization: yes**

Figure 4.3 Component Inspector — Enabled Interrupt



Name	Value	Details
Module name	PIT	PIT
Counter	PIT_CVAL0	PIT_CVAL0
Counter direction	Down	
Counter width	32 bits	
Value type	Optimal	uint32_t
Input clock source	Internal	
Counter frequency	20.972 kHz	Timing setting failed - it is impossible ...
Counter restart	On-match	
Period device	PIT_LDVAL0	PIT_LDVAL0
Period		Unassigned timing
Interrupt	Enabled	
Interrupt priority	medium priority	8
Channel list	0	

For code generation, refer to the Code Generation topic.

Code Generation

To generate code, in the Project Explorer window, select **ProcessorExpert.pe** and right-click on it. Select **Generate Processor Expert Code** option from the context menu. This process generates source files for components to the **Generated_Code** folder in the CodeWarrior project window. The other modules can be found in the **Sources** folder.

Writing Application Code

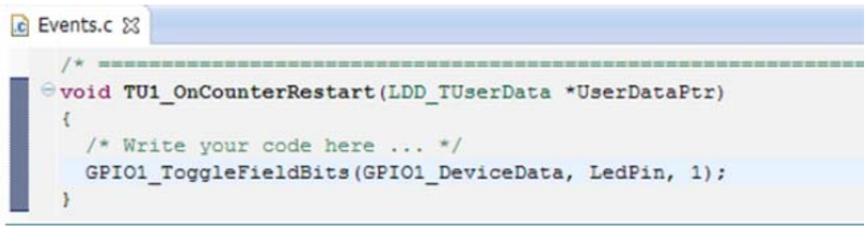
In the **Components** view, expand the **TU1: TimerUnit_LDD** component's list of events and methods. To view the source code of the **TU1_OnCounterRestart** in editor, select it and right-click on it to select **View Code** option from the context menu.

Insert the following lines into the TU1_OnCounterRestart event handler function body:

```
GPIO1_ToggleFieldBits(GPIO1_DeviceData, LedPin, 1);
```

NOTE The first parameter of LDD component methods is the `LDD_TDeviceData` structure pointer. If the component is set to **Auto initialization: yes** as in this example, then there is generated a `{component_name}_DeviceData` constant that can be used. If not, it is up to the user to call the `Init` method first and then use the returned pointer as the first parameter.

Figure 4.4 Inserted Command



```
/* ============================================================
void TU1_OnCounterRestart(LDD_TUserData *UserDataPtr)
{
  /* Write your code here ... */
  GPIO1_ToggleFieldBits(GPIO1_DeviceData, LedPin, 1);
}
```

Running the Application

1. Build the project using the **Project > Build** command from the CodeWarrior menu.
2. If you have the TWR–K60N512 board, connect it now. Run and debug the application using the **Run > Run** command from the CodeWarrior menu. When the application starts running, one of the the on-board LEDs start blinking.

Index

Symbols

"PESL".h 100
__DI() 103
__EI() 103
__initialize_hardware() 97
_DEVICE 122
_startup() 97
{projectname}_SIGNALS.doc 83
{projectname}_SIGNALS.txt 83
{startupfile}.c 100

Numerics

16-bit Register 103

A

A/D converter 13
Abstraction 55
Access Macros 123
Advanced view 39
AfterNewSpeed 110
Alphabetical 28
Ancestor 84
Assistant 28

B

Bare-metal Applications 58
Basic view 38
BeforeNewSpeed 110
bit 48
bits 48
BitsIO 116
buffer overflow 15
Bus clock 15
ByteIO 116

C

Code Generation 27
Code generation log 99
Code generator 10
ColdFire+ 76
Communication Components 118
Component 15
Component Assistant 29
Component Categories 28
Component Driver 15
Component Enabled 27
Component icon 52
Component Inspector 15, 39
Component Inspector View 30
Component module 98
Component sharing 85
Components 125
Components Library 28
Configuration registers 27
PEcfg_ 77
ConnectPin 86
ConnectPin Method 86
Constants 102
Counter 15
CPU.H 78
Creating Applications 14

D

Deinit() 58
Descendant 84
design specifications 10
Design-time verifications 9
DisableEvent 110
Distinct mode 27
Documentation 125

E

Easy Initialization 52
Embedded Components 13
EnableEvent 110
EnterCritical() 103
ERR_ARBITR 105
ERR_BREAK 105
ERR_BUSOFF 104
ERR_BUSY 104
ERR_COMMON 105
ERR_CRC 105

ERR_DISABLED 104
ERR_ENABLED 104
ERR_FAILED 105
ERR_FAULT 104
ERR_FRAMING 104
ERR_IDLE 104
ERR_LINSYNC 105
ERR_MATH 104
ERR_NOISE 104
ERR_NOTAVAIL 104
ERR_OK 104
ERR_OVERFLOW 104
ERR_OVERRUN 104
ERR_PARITY 104
ERR_PROTECT 105
ERR_QFULL 105
ERR_RANGE 104
ERR_RXEMPTY 104
ERR_SPEED 104
ERR_TXFULL 104
ERR_UNDERFLOW 105
ERR_UNDERRUN 105
ERR_VALUE 104
Event module 99
Events 14, 15
ExitCritical() 103
Extensible components library 9
Extensible Library 54
External Devices 54
External user module 15
External Xtal frequency 64

F
Filtering 28
fixed value 44
FPGA 8
FPGAs 52
Free running device 15
from interval 45

G
Generated_Code 125
getReg{w}BitGroupVal 124
Graphical IDE 10

H
Help on Component 39
Help on component 27
Hertz 48
Hide views 20
High level component 15
High Level Components 55
High speed mode 64
Higher visibility level 38
Hints 51
Hz 48

I
Images 103
Import package 20
Inheritance 84
Init 111
Init() 58
Inspector 26
Inspector - Pinned 26
Inspector Items 35
Internal Peripherals 54
Internal peripherals 16
Interrupt Priority 75, 79
Interrupt Service Routine 73
Interrupt Subroutines 54
Interrupt Vector Table 73
Interrupts 73
Interrupts and Events 73
IO_Map.c 100
IO_Map.h 100, 123
ISR 16, 54
ISRs 24
Items Visibility 38

K
kbit 48
kbits 48
kHz 48
kilohertz 48
Kinetis 76

L
LCF 58
LDD 9, 55
LDD Components 55
LDD components 9, 15, 16
Logical Device Drivers 9, 55
Logical Device Drivers) 15
Low level component 16
Low Level Components 56
Low Power Features 59
Low speed mode 64

M
Macros 102, 103
Main module 99
megahertz 48
Memory 79
Methods 16
methods 14
MHz 48
Microcontroller 16
Microcontroller Unit 16
microseconds 47
milliseconds 47
MK60X256VMD100 141
Modes 28
Module 16
ms 47

N
Named Bits 123
Named Groups of Bits 123

O
On-Chip Peripherals 78
On-chip Peripherals 52
On-chip Peripherals Management 53
OnReset 97
OOP 16
Open File 27

P
PDD 120
PE 16
PE_Const.h 100
PE_Error.h 100
PE_low_level_init() 97
PE_Timer 100
PE_Types.h 100, 123
Peripheral Initialization component 16
Peripheral Initialization Components 56
PESL 16, 122
PESL commands 24
Physical Device Drivers 120
Pin hint 51
Pin_signal 82
Pins 51
PLL 16, 64
Pop-up Menu 28
Popup menu 16
Portability 61
PPG 46, 52
Predefined Types 102
Preferences 21
Prescaler 17
Priority of Event Code 76
Processor 17
Processor Component 15
Processor module 98
Processor ticks 47
Processors 28
Project Options 20
Project Pop-up Menu 20
Properties 17
PWM 46

R
Read Only Items 33
Remove component from project 27
Resources Allocation 85
RTOS 17, 58
RTOS Adapter 58
RTOS environment 58

S
Save Component Settings 27
second 48

seconds 47
selectable visibility 38
setReg{w}BitGroupVal 124
Shared Ancestor 84
Shared Drivers 59
Shared mode 27
Shared modules 100
Shared Pins 52
Sharing Pins 85
Show views 20
Signal names 99
Slow speed mode 64
Sources 125
speed mode tabs 47
Speed Modes 81, 116
sub-clock xtal frequency 64
SW 55
system behavior 9

T

Template 17
Templates 125
testReg{w}BitGroup 123
ticks 47
Timer ticks 47
TimerX 52
TimerX_PPG 52
TimerX_PWM 52
Timing 79
Timing Model 80
Timing Precision Configuration 45
Timing Settings 43, 79
Timing Values Table 45
TWR–K60N512 141
Types 102

U

Unused Methods 115
us 47
User module 17
User-defined Component Template 17

V

Vectors.c 100
View Code 27
View Menu 33
View Mode Buttons 33

X

XML documentation 99
Xtal 17