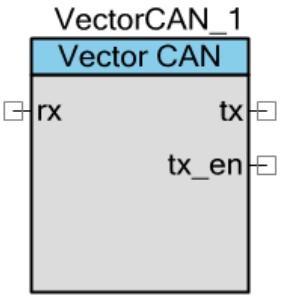
Vector CAN
1.0



Features

- CAN2.0 A/B protocol implementation, ISO 11898-1 compliant
- Programmable bit rate up to 1 Mbps @ 8 MHz (BUS_CLK)
- Two or three wire interface to external transceiver (Tx, Rx, and Tx Enable)
- Driver provided and supported by Vector

General Description

The Vector CANbedded environment consists of a number of adaptive source code components that cover the basic communication and diagnostic requirements in automotive applications.

The Vector CANbedded software suite is customer specific and its operation will vary according to application and OEM. This component for the Vector CANbedded suite is written to generically support the CANbedded structure regardless of the flavor of the particular OEM application.

The Vector CAN component developed for PSoC3 allows easy integration of the Vector certified CAN driver.

When to Use a Vector CAN

The Vector CAN component is used when you need integration with a CAN driver for PSoC 3 provided by Vector.

Input/Output Connections

This section describes the various input and output connections for the Vector CAN component. An asterisk (*) in the list of I/Os indicates that the I/O may be hidden on the symbol under the conditions listed in the description of that I/O.

rx – Input

CAN bus receive signal (connected to CAN RX bus of external transceiver).

tx – Output

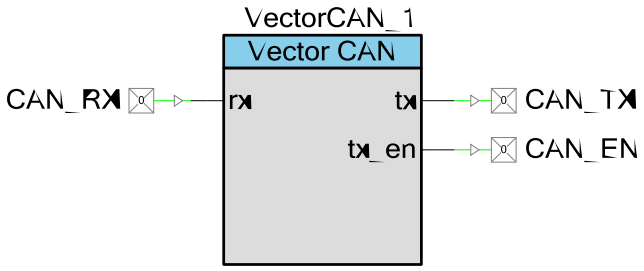
CAN bus transmit signal, (connected to CAN TX bus of external transceiver).

tx_en – Output *

External transceiver enable signal. This output displays when the **Add Transceiver Enable Signal** option is selected in the **Configure** dialog.

Schematic Macro Information

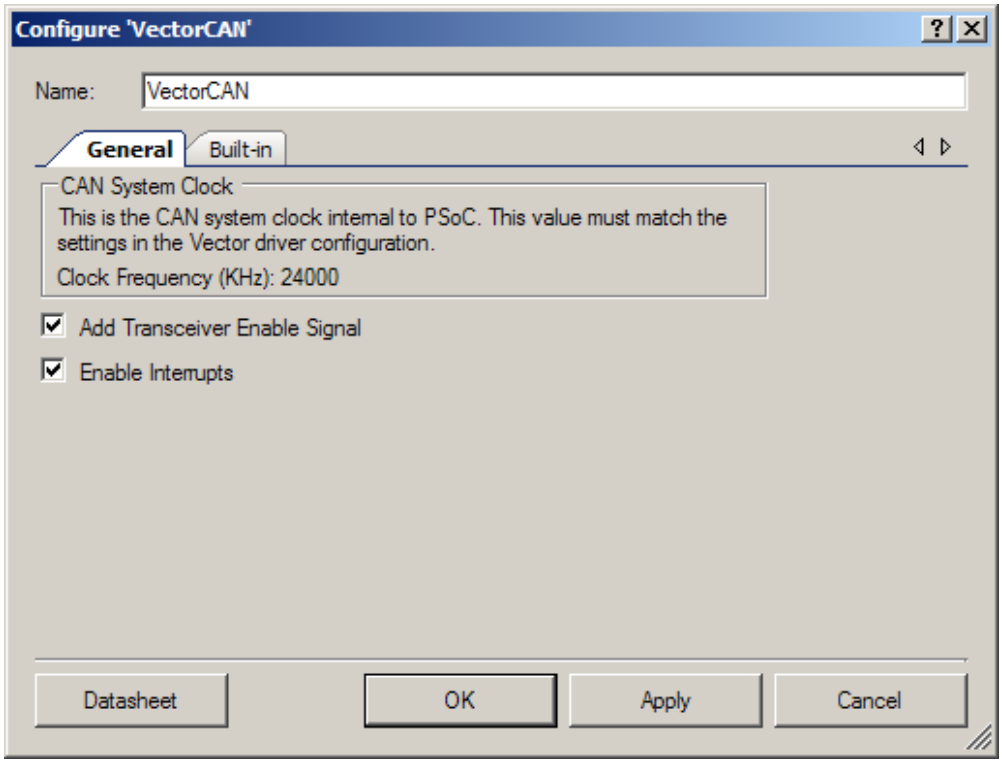
The default Vector CAN in the Component Catalog is a schematic macro using a Vector CAN component with default settings. The Vector CAN component is connected to Input and Output Pin components. The Pins components are also configured with default settings, except that Input Synchronized is set to false in the Input Pin component.



Component Parameters

Drag a Vector CAN component onto your design and double click it to open the **Configure** dialog. This dialog has a **General** tab to guide you through the process of setting up the Vector CAN component.

Figure 1. Configure Vector CAN Dialog



The **General** tab provides the following parameters.

Add Transceiver Enable Signal

Enables/disables the use of the tx_en signal for the external CAN transceiver. The default setting is **Enable**.

Enable Interrupts

Enables/disables global interrupts from the CAN. The default setting is **Enable**. Should be disabled if the driver files are configured for polling mode (otherwise compilation errors appear).

Clock Selection

The Vector CAN component is connected to the BUS_CLK clock signal. A minimum value of 8 MHz is required to support all standard CAN baud rates up to 1 Mbps. The value of the BUS_CLK selected in the PSoC 3 project design-wide resources must be the same as the value selected in the Vector CAN driver configuration for bus timing.

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent sections cover each function in more detail.

By default, PSoC Creator assigns the instance name “Vector_CAN_1” to the first instance of a component in a given design. You can rename the instance to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is “Vector_CAN.”

Function	Description
Vector_CAN_Start()	Initializes and enables the Vector CAN component using the Vector_CAN_Init() and Vector_CAN_Enable() functions.
Vector_CAN_Stop()	Disables the Vector CAN component.
Vector_CAN_GlobalIntEnable()	Enables Global Interrupts from CAN Core.
Vector_CAN_GlobalIntDisable()	Disables Global Interrupts from CAN Core.
Vector_CAN_Sleep()	Prepares the component for sleep.
Vector_CAN_Wakeup()	Restores the component to the state when Vector_CAN_Sleep() was called.
Vector_CAN_Init()	Initializes the Vector CAN component based on settings in the component customizer. Sets up the CAN interrupt with the interrupt service routine CanIsr_0() generated by the Vector CAN configuration tool.
Vector_CAN_Enable()	Enables the Vector CAN component.
Vector_CAN_SaveConfig()	Saves the component configuration.
Vector_CAN_RestoreConfig()	Restores the component configuration.

Global Variables

Variable	Description
Vector_CAN_initVar	Vector_CAN_initVar indicates whether the Vector CAN has been initialized. The variable is initialized to 0 and set to 1 the first time Vector_CAN_Start() is called. This allows the component to restart without reinitialization after the first call to the Vector_CAN_Start() routine. If reinitialization of the component is required, then the Vector_CAN_Init() function can be called before the Vector_CAN_Start() or Vector_CAN_Enable() function.

uint8 Vector_CAN_Start(void)

Description: This is the preferred method to begin component operation. Vector_CAN_Start() sets the initVar variable, calls the Vector_CAN_Init() function, and then calls the Vector_CAN_Enable() function.

Parameters: None

Return Value: Indication whether register is written and verified.

Side Effects: None

uint8 Vector_CAN_Stop(void)

Description: Disables the Vector CAN component.

Parameters: None

Return Value: Indication whether register is written and verified.

Side Effects: None

uint8 Vector_CAN_GlobalIntEnable(void)

Description: This function enables global interrupts from the CAN Core.

Parameters: None

Return Value: Indication whether register is written and verified.

Side Effects: None

uint8 Vector_CAN_GlobalIntDisable(void)

Description: This function disables global interrupts from the CAN Core.

Parameters: None

Return Value: Indication whether register is written and verified.

Side Effects: None

void Vector_CAN_Sleep(void)

Description: This is the preferred routine to prepare the component for sleep. The Vector_CAN_Sleep() routine saves the current component state. Then it calls the Vector_CAN_SaveConfig() function and calls Vector_CAN_Stop() to save the hardware configuration.

Call the Vector_CAN_Sleep() function before calling the CyPmSleep() or CyPmHibernate() functions.

Parameters: None

Return Value: None

Side Effects: None

void Vector_CAN_Wakeup(void)

Description: This is the preferred routine to restore the component to the state when Vector_CAN_Sleep() was called. The Vector_CAN_Wakeup() function calls the Vector_CAN_RestoreConfig() function to restore the configuration. If the component was enabled before the Vector_CAN_Sleep() function was called, the Vector_CAN_Wakeup() function will also re-enable the component.

Parameters: None

Return Value: None

Side Effects: Calling the Vector_CAN_Wakeup() function without first calling the Vector_CAN_Sleep() or Vector_CAN_SaveConfig() function may produce unexpected behavior.

void Vector_CAN_Init (void)

Description: Initializes or restores the component according to the customizer Configure dialog settings. It is not necessary to call Vector_CAN_Init() because the Vector_CAN_Start() routine calls this function and is the preferred method to begin component operation. This function sets up the CAN interrupt with the interrupt service routine CanIsr_0() generated by the Vector CAN configuration tool.

Parameters: None

Return Value: None

Side Effects: None

uint8 Vector_CAN_Enable(void)

Description: Activates the hardware and begins component operation. It is not necessary to call Vector_CAN_Enable() because the Vector_CAN_Start() routine calls this function, which is the preferred method to begin component operation.

Parameters: None

Return Value: Indication whether register is written and verified.

Side Effects: None

void Vector_CAN_SaveConfig(void)

Description: This function saves the component configuration. This will save nonretention registers. This function will also save the current component parameter values, as defined in the Configure dialog or as modified by appropriate APIs. This function is called by the Vector_CAN_Sleep() function.

Parameters: None

Return Value: None

Side Effects: None

void Vector_CAN_RestoreConfig(void)

Description: This function restores the component configuration. This will restore nonretention registers. This function will also restore the component parameter values to what they were prior to calling the Vector_CAN_Sleep() function.

Parameters: None

Return Value: None

Side Effects: Calling this function without first calling the Vector_CAN_Sleep() or Vector_CAN_SaveConfig() function may produce unexpected behavior.

Sample Firmware Source Code

PSoC Creator provides numerous example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

Refer to the “Find Example Project” topic in the PSoC Creator Help for more information.

Interrupt Service Routines

The Vector driver uses the CAN interrupt, allowing you access to it. The Vector_CAN_Init() function sets up the CAN interrupt with the interrupt service routine CanIsr_0() generated by the Vector CAN configuration tool.

Functional Description

For a complete description of the CAN Core, refer to the Controller Area Network (CAN) chapter in the *PSoC 3 and PSoC 5 Technical Reference Manual*.

For a complete description of the Vector GENy tool, refer to the Vector GENy tool documentation.

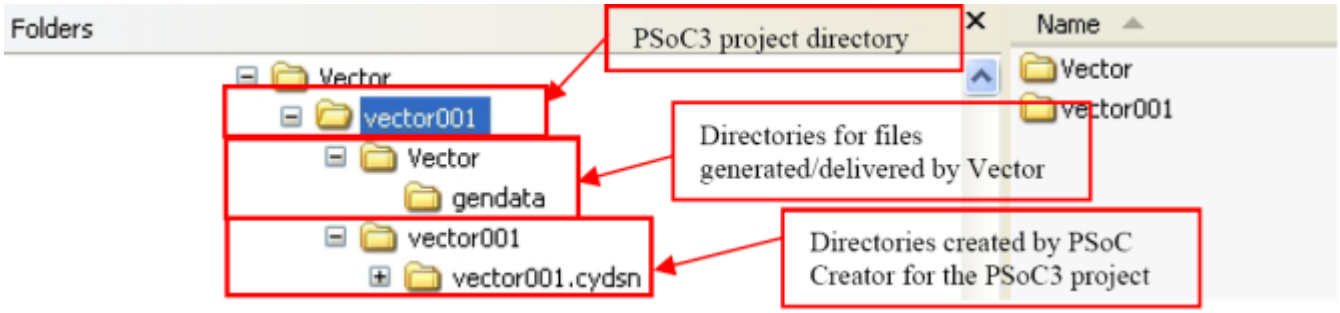
Creating a Project with the Vector GENy Tool

1. Create a Blank PSoC Project and a Directory for the CAN Driver Files

Before starting to work with the Vector GENy tool to generate the CAN driver files, you should create a PSoC project, so that the Vector GENy tool can place the driver files directly in the PSoC project directory. The PSoC project will be empty for the moment.

Create a directory for the CAN driver generated files in the PSoC project directory, as shown in Figure 2.

Figure 2. PSoC Project Directory



2. Generate the Vector CAN Driver Files for the PSoC Application

The Vector GENy tool generates the CAN driver files for PSoC based on:

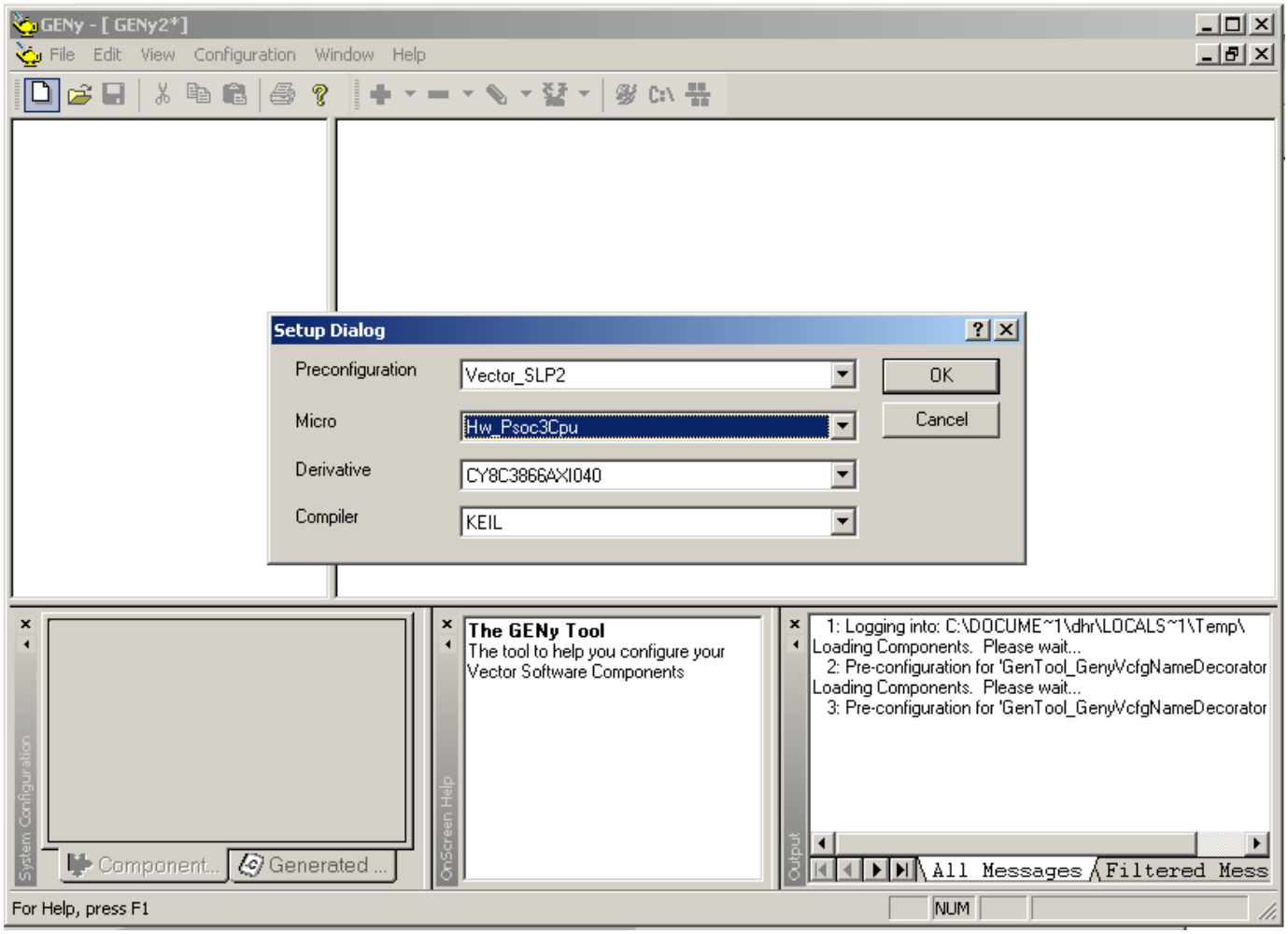
- The CAN application message database
- The configuration in Vector GENy

After you load the database, you can configure the CAN driver generation tool (GENy) to generate a driver to handle Vector CAN messages.

3. Open the Vector GENy Tool and Create a New Configuration

1. Start the Vector GENy tool from Start > All Programs > Vector GENy 1.4 >GENy.
2. Click on the **New** button (see Figure 3).

Figure 3. Vector GENy Tool after Clicking on the "New" Button



The **Setup Dialog** that is displayed when you click on the **New** button has no options other than the ones shown, so click **OK.**

4. Set Up the Configuration Before Generating CAN Driver Files

After the database is loaded, configure the PSoC driver with the following (this is just a "getting started" configuration):

1. Select the PSoC CAN module component.
2. Select the Tx and Rx messages that the driver will use.

3. Select the bus timing and message acceptance filter.
4. Select options for polling or interrupt mode.
5. Select the directories where the generated files will be placed.
6. After selecting the Tx and Rx messages, select and configure the acceptance filter and bus timing as shown in the figures below.

The following figures show these steps.

Figure 4. Acceptance Filter and Bus Timing

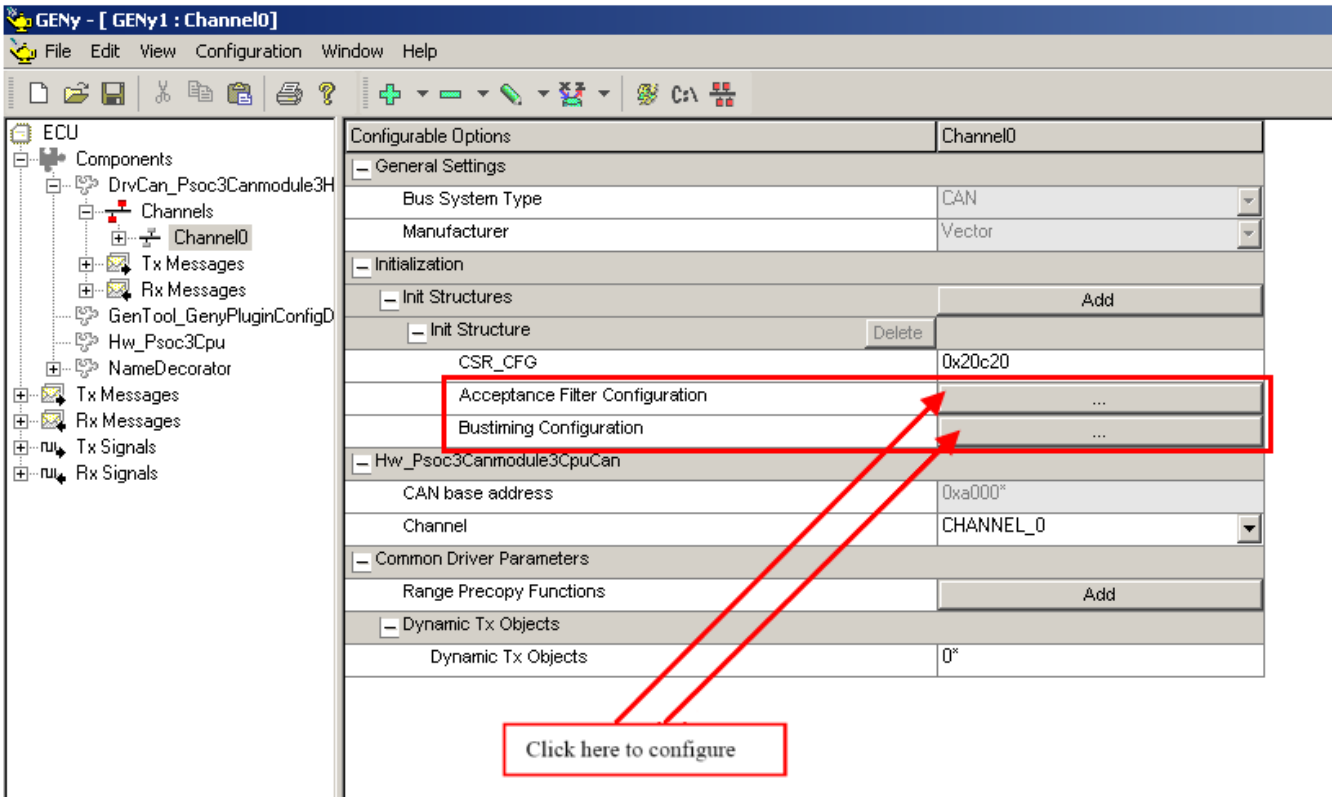
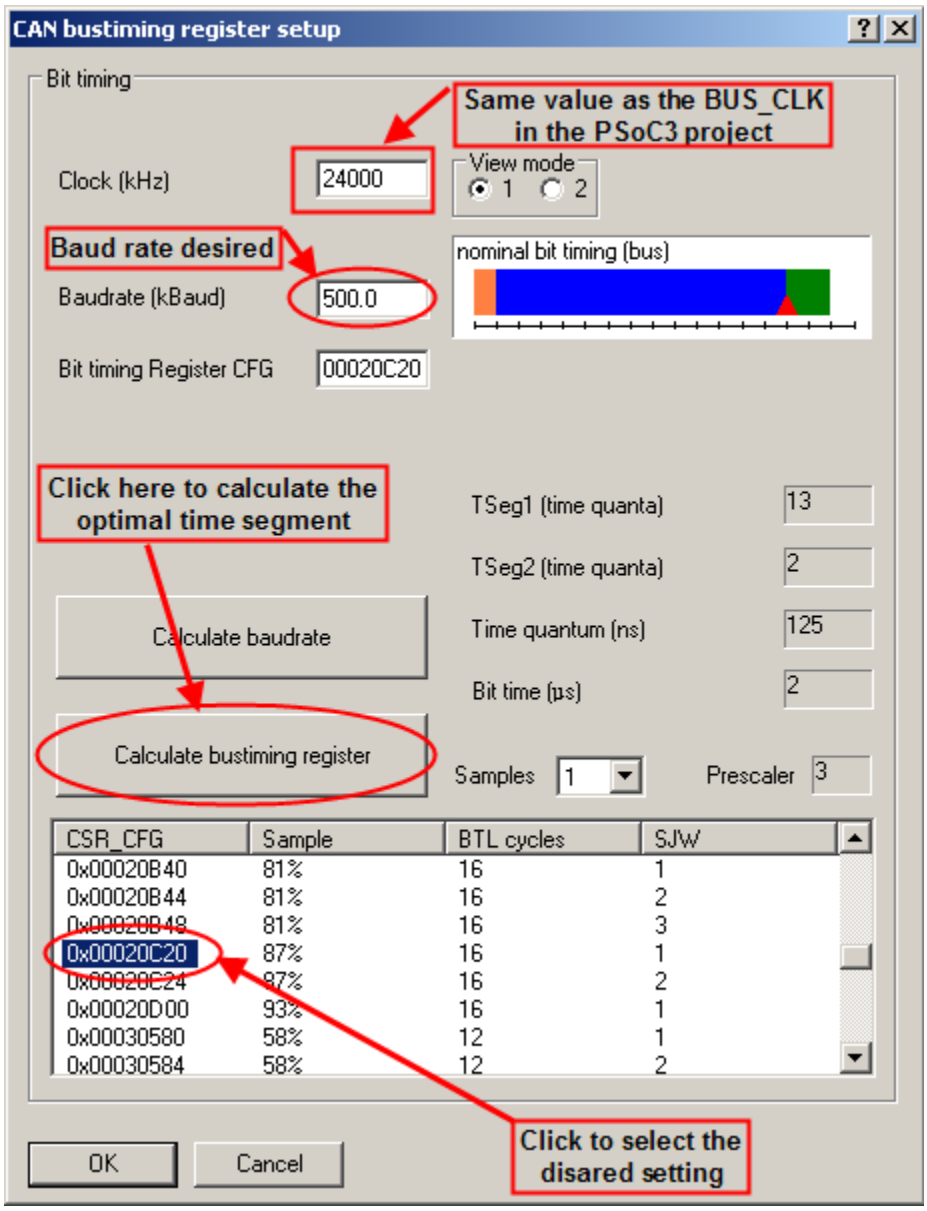


Figure 5. Bus Timing Setting



The clock selected in the **CAN bustiming** window must be the same as BUS_CLK in the PSoC 3 project.

When choosing polling or interrupt mode, remember that in polling mode the application has to call the function CanTask() to service pending events or messages. In interrupt mode, events or messages are serviced by the interrupt service routine CanIsr_0().

After the acceptance filter and bus timing and mailbox polling/interrupt modes are set up, select the CAN driver files directory.

Create these directories within the PSoC project directory (as described earlier) so that everything related to the project is in one place.

Figure 6. Selecting the Polling or Interrupt Mode

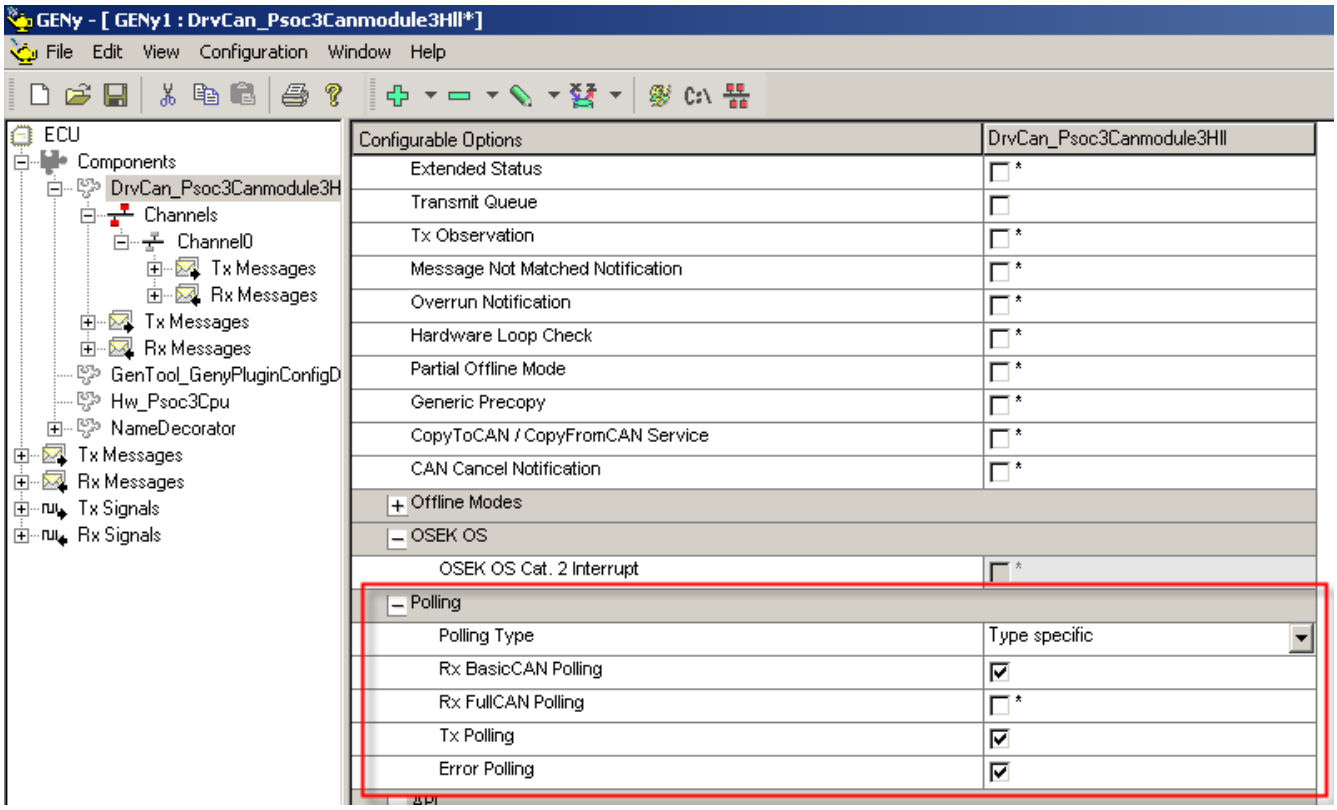


Figure 7. Selecting the Destination Directories

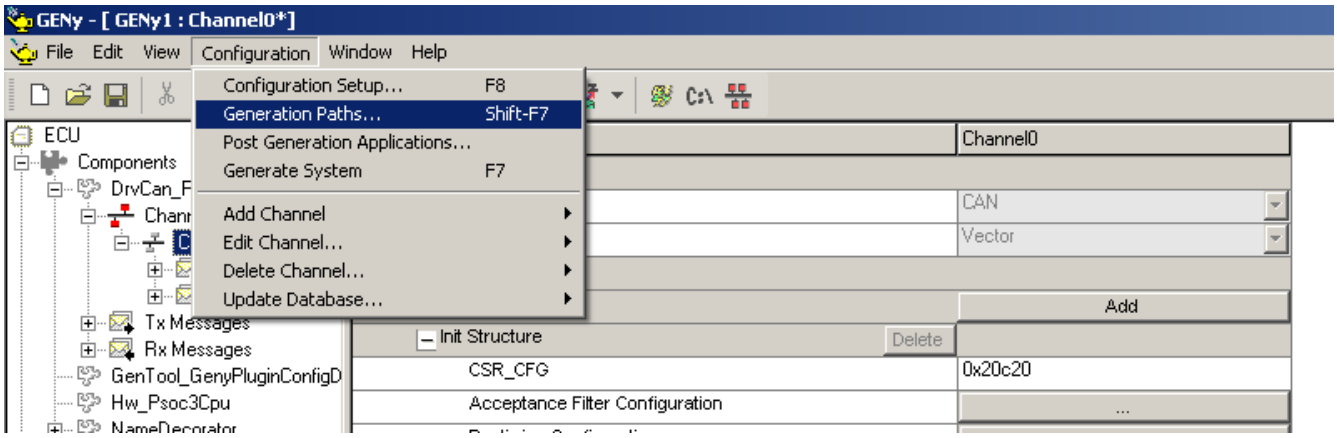
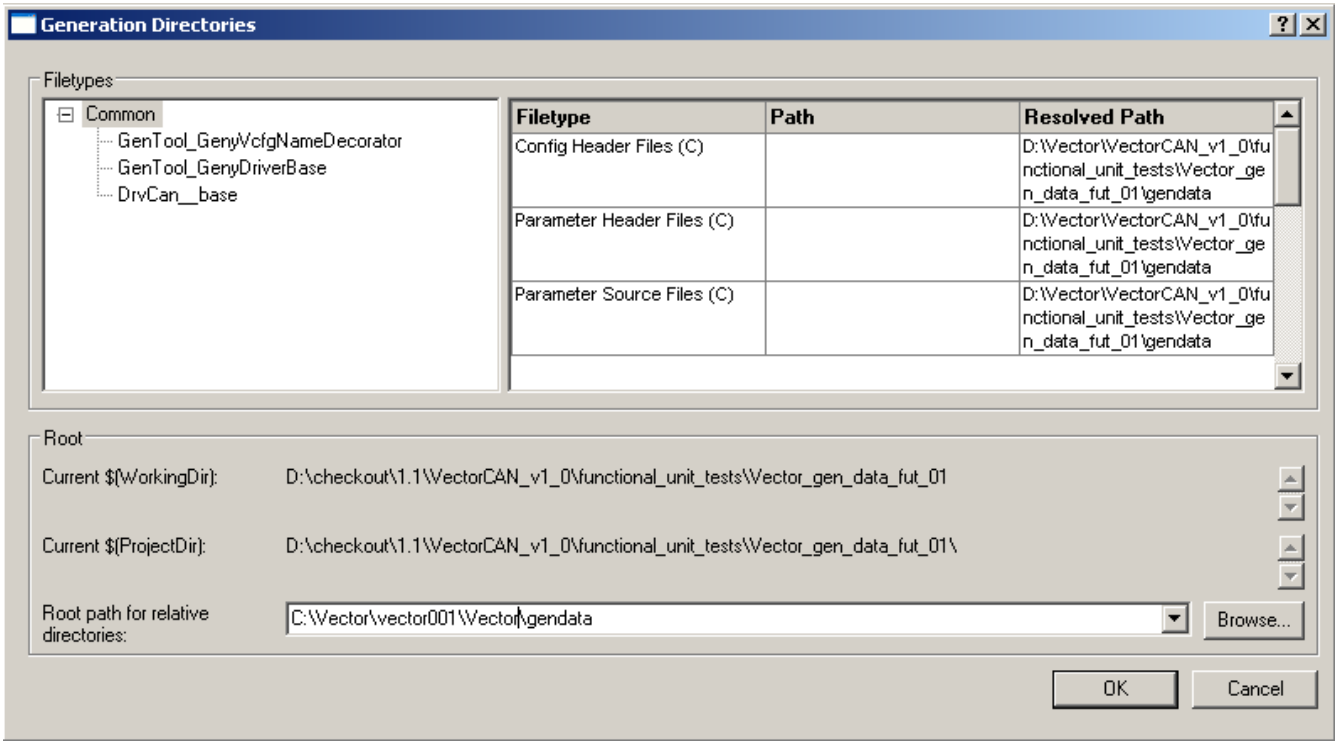


Figure 8. Selecting the Root Path for the Relative Directories



In the **Generation Directories** dialog window, set **Root path for relative directories** to the subdirectory created in the PSoC project directory.

5. Generate the CAN Driver Files

At this point, the tool is ready to generate the PSoC driver files, which will be placed in the selected directory created in the PSoC project.

You should also save the configuration file in the PSoC project directory, so that when modifications are needed and new files are generated, everything is located in the same project directory.

These files are generated every time the **Generate System** button is pressed, but there are also some common files (that do not change) which should be copied in the same PSoC project directory for convenience.

The following figures illustrate the final steps before moving into PSoC Creator to build the project.

Figure 9. Click on "Generate System" to Generate the Files in the PSoC Project Subdirectory

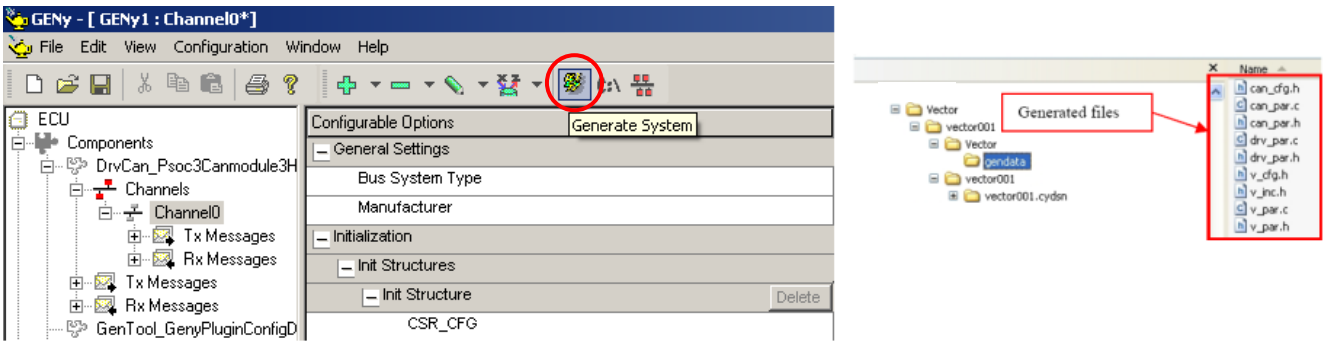
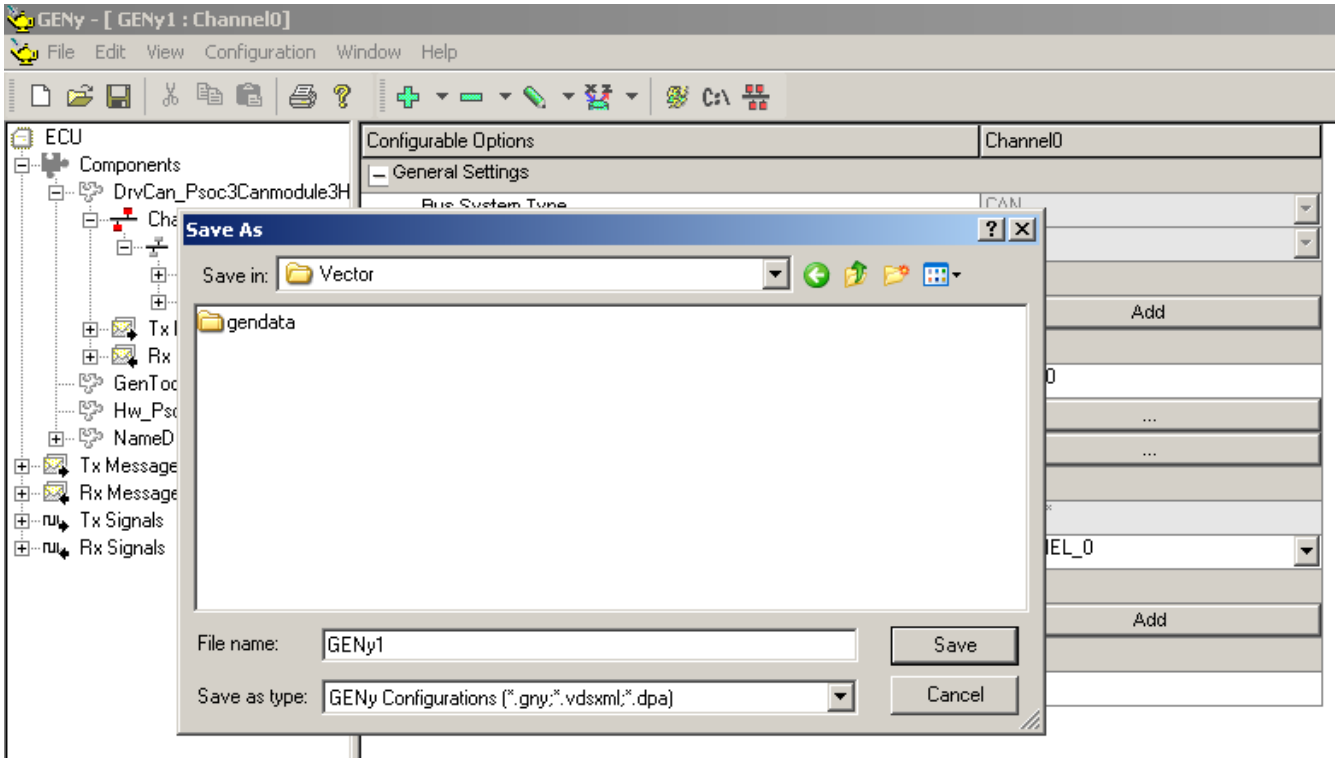


Figure 10. Save the Vector GENy Configuration in the PSoC Project Subdirectory

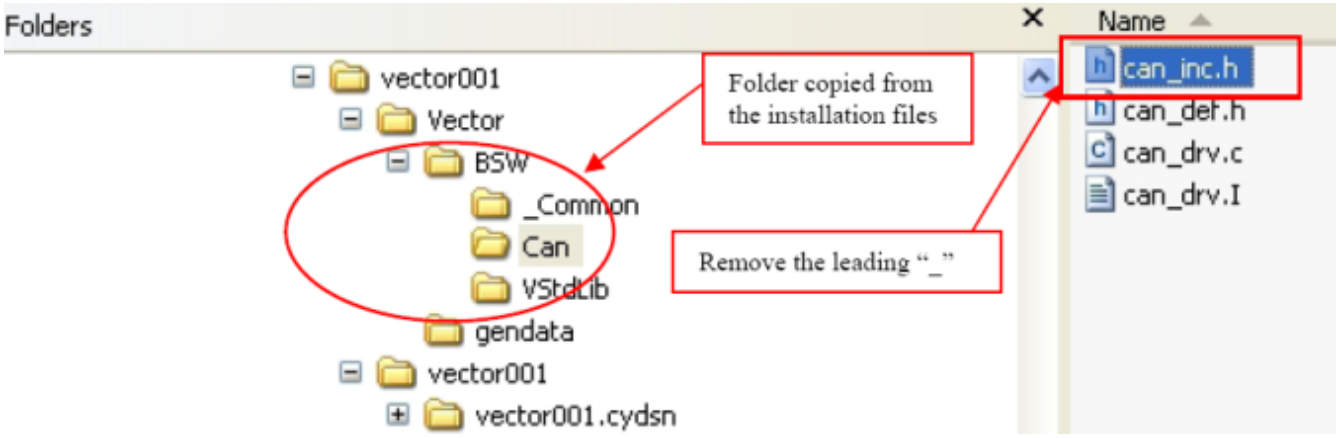


If, during installation, the default directories were accepted, the common files are located in:

`C:\Vector\CBD0810092_D00_Psoc3\BSW`

The full BSW directory should be copied into the PSoC project folder, and the *_can_inc.h* file name in the Can subdirectory should be changed to *can_inc.h* (remove the leading "_").

Figure 11. CAN Driver Common Directories Copied to the PSoC Project Folder



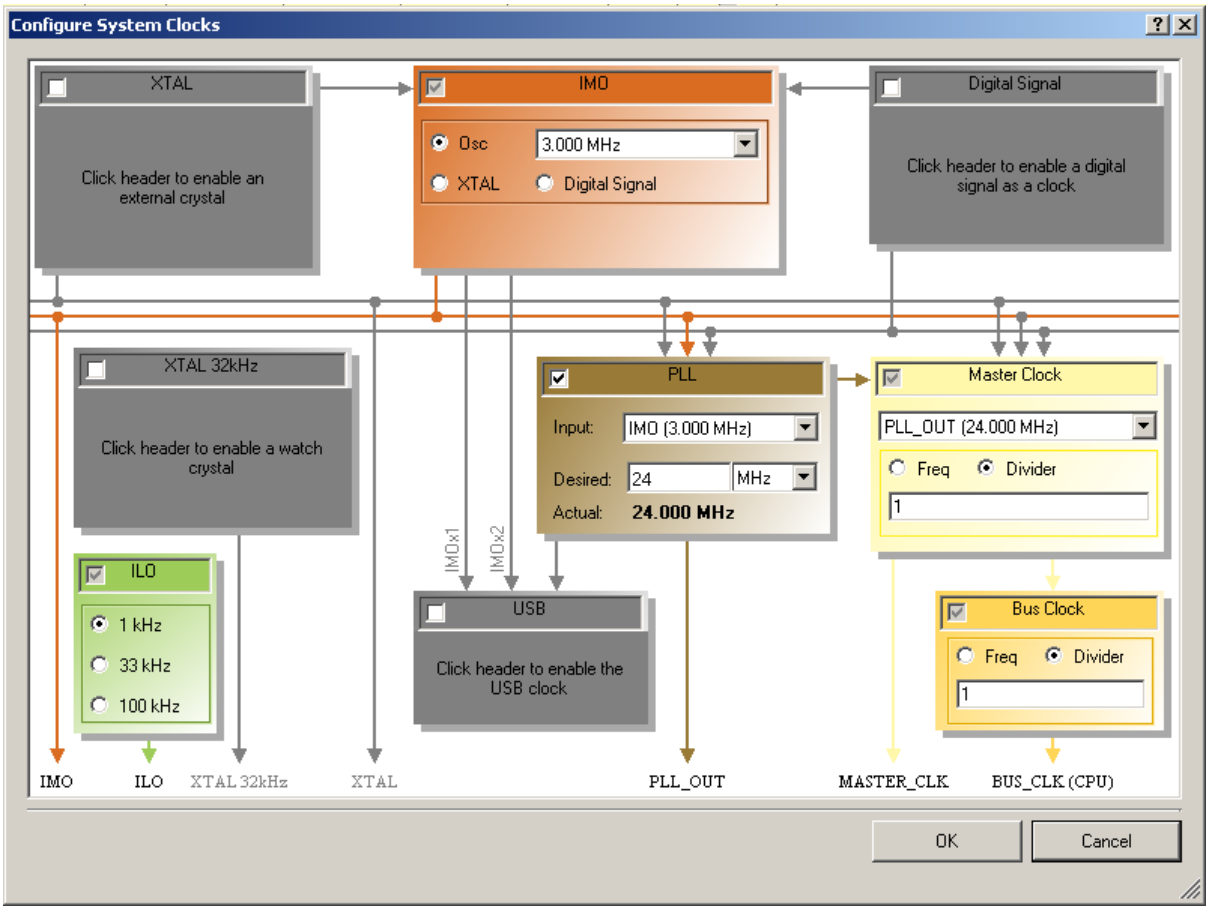
Now you can close the Vector GENy tool. Everything is set up to proceed with the PSoC project.

6. Create the PSoC Project

1. Place a Vector CAN component into the schematic and open the configure dialog to customize component options, enabling interrupts and using tx_en output.
2. Setup the clock tree.
3. Place the pins.
4. Import Vector CAN driver files.
5. Change the "Build settings…" to include the CAN driver directories and set NOOVERLAY option.
6. Write application in *main.c*.

Set Up the Clock Tree

Figure 12. Clock Setup



Place the Pins

In the example shown in Figure 13, the pins are placed to be used with the CAN/LIN EBK (CY8CKIT-017) for convenience. You can use any pin.

Figure 13. Pin Selection

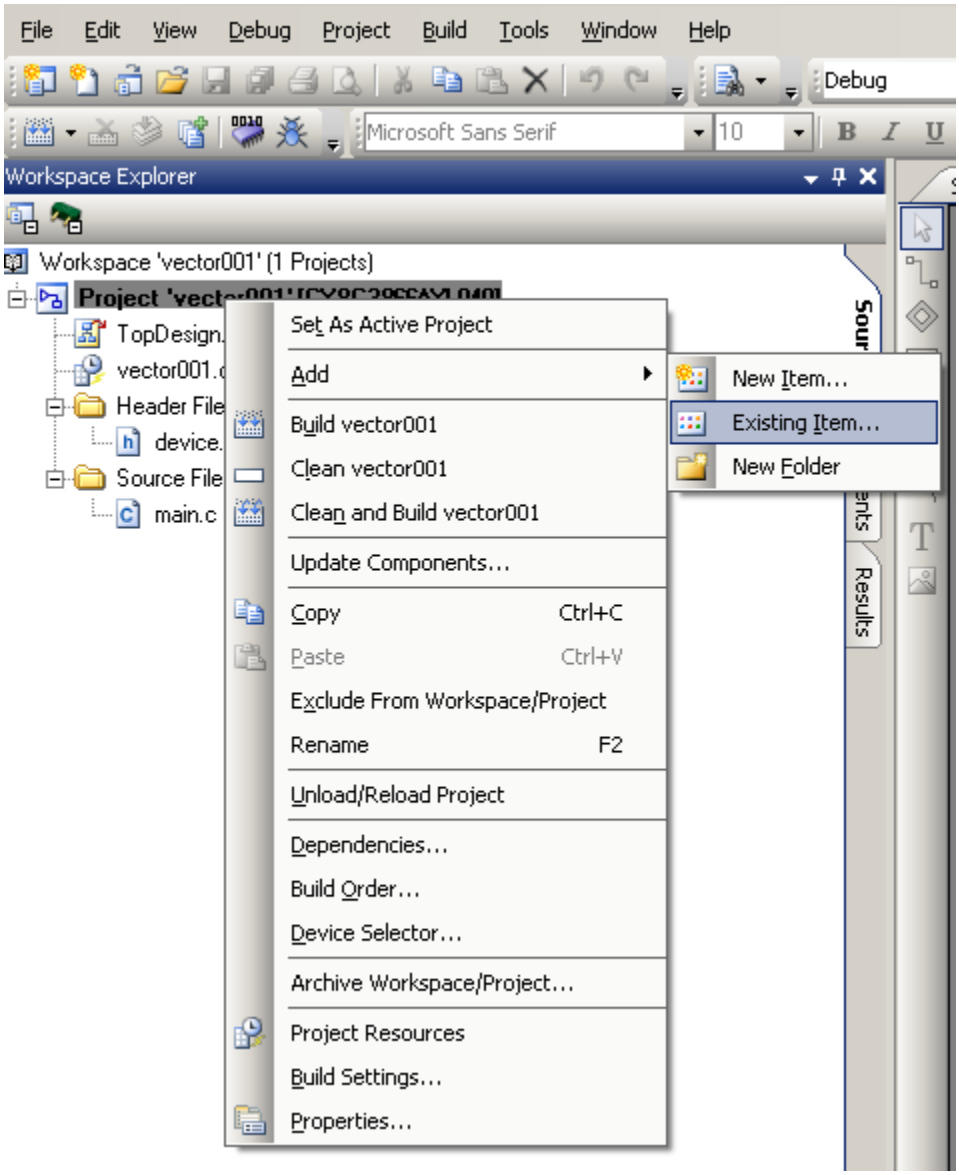
Alias	Name	Pin	Lock
	Pin_tx_en	P3[1]	☑
	Pin_tx	P3[3]	☑
	Pin_rx	P3[2]	☑

Import Vector CAN Driver Files

Both header files and source files must be imported from the “gendata” directory and from all folders under “BSW.”

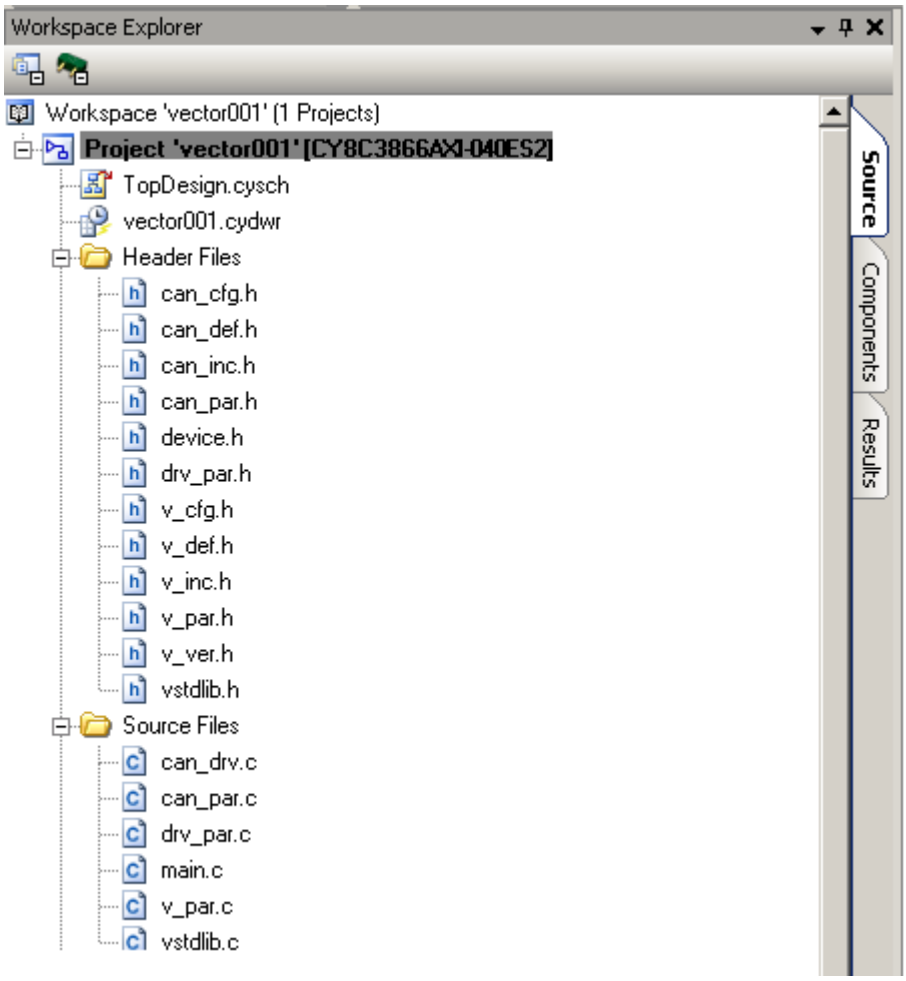
To import files, right click on **Header Files** and **Source Files** in the PSoC Creator workspace explorer, and select the menu to add existing items. Repeat the process until all Vector files are imported.

Figure 14. Importing Existing Header and Source Files



When the import step is finished, the workspace explorer should look like Figure 15.

Figure 15. Files List after Importing



Change the “Build settings…” to Include the CAN Driver Directories

Figure 16. Add Include Directories

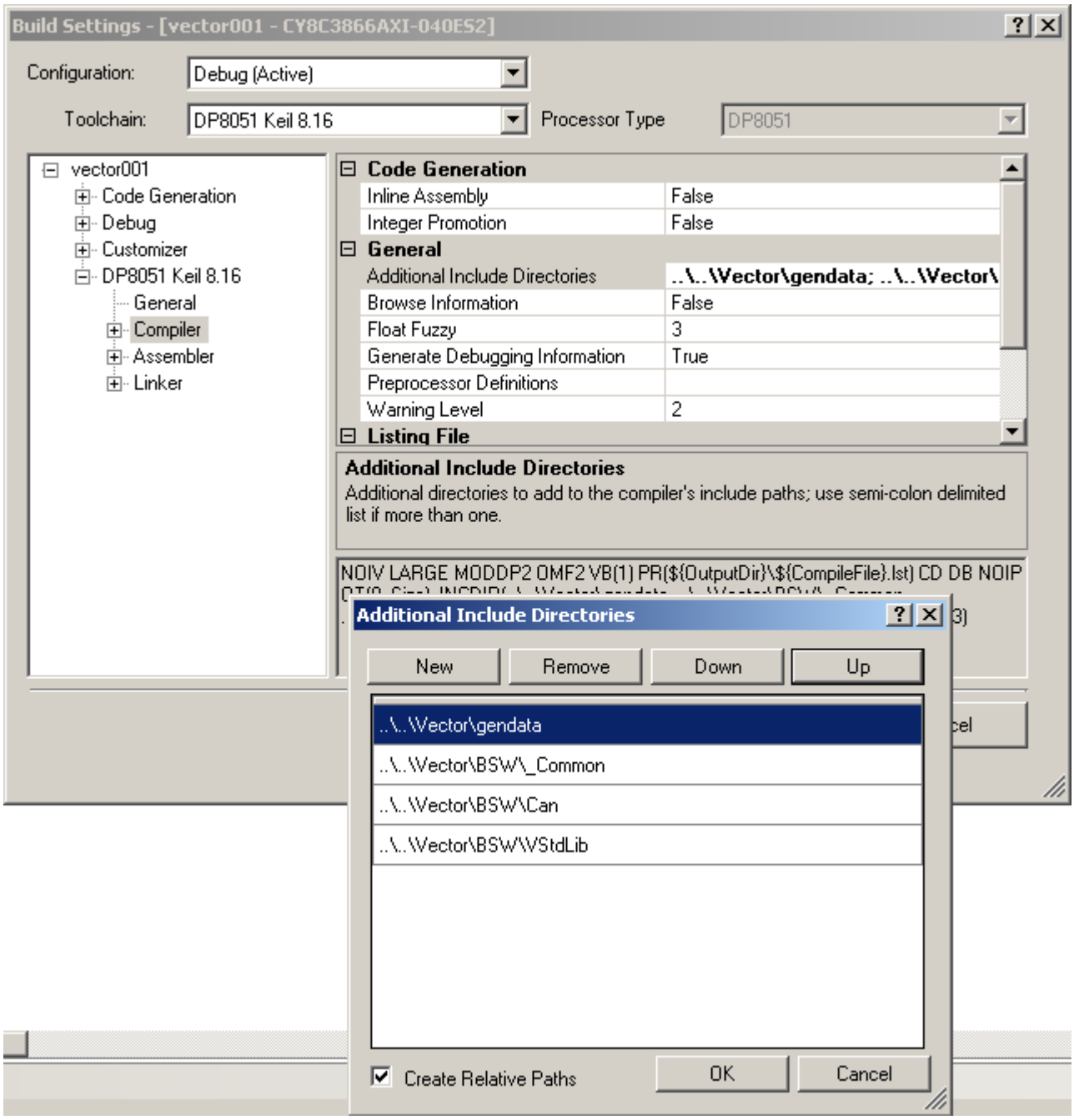
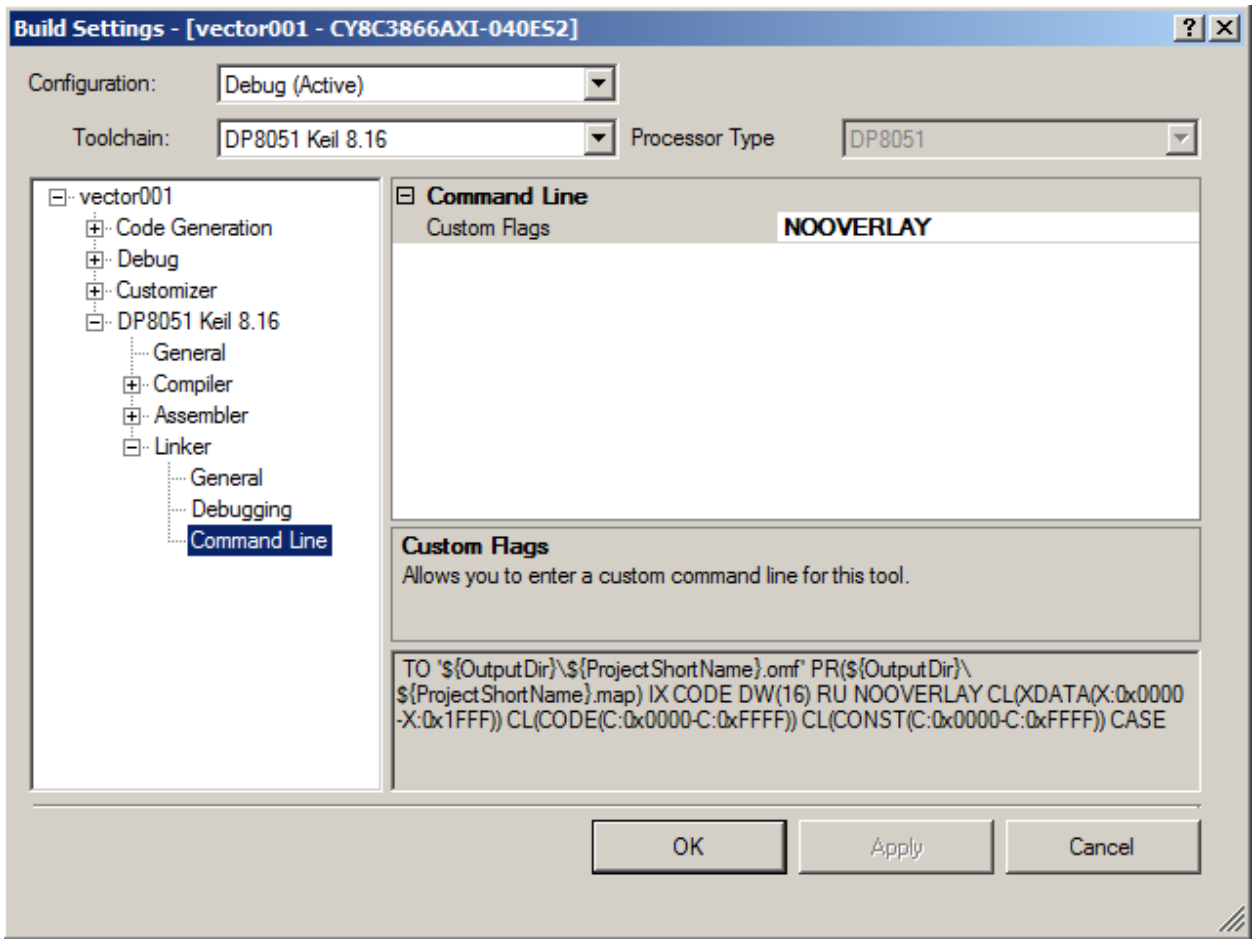


Figure 17. Set NOOVERLAY option



The Vector CAN Driver APIs use function pointers. The Keil compiler for PSoC 3 does function call analysis to determine how it can overlay function variables and arguments. When function pointers are present the compiler cannot adequately analyze the calling structure, so the NOOVERLAY option is selected to avoid problems that occur because of the use of function pointers. Further information on the handling of function pointers with the Keil compiler is available in the application note: Function Pointers in C51 (www.keil.com/appnotes/docs/apnt_129.asp).

In main the initialization process requires you to:

- Include the *v_inc.h* file for the driver in main.c.
- Enable global interrupts if required.
- Call the Vector_CAN_Start() function.
- Call the CanInitPowerOn() function (generated by the Vector GENy tool).
- Write the necessary functionality using an API from Vector CAN and generated by the Vector GENy tool.

Resources

The Vector CAN component uses the dedicated CAN hardware block in the silicon.

API Memory Usage

The component memory usage varies significantly, depending on the compiler, device, number of APIs used and component configuration. The following table provides the memory usage for all APIs available in the given component configuration.

The measurements have been done with the associated compiler configured in Release mode with optimization set for Size. For a specific design the map file generated by the compiler can be analyzed to determine the memory usage.

Configuration	PSoC 3 (Keil_PK51)		PSoC 5 (GCC)		PSoC 5LP (GCC)	
	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
Default	602	18	N/A	N/A	N/A	N/A

DC and AC Electrical Characteristics

Specifications are valid for –40 °C ≤ TA ≤ 85 °C and TJ ≤ 100 °C, except where noted.
Specifications are valid for 1.71 V to 5.5 V, except where noted.

CAN DC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
I_{DD}	Block current consumption		--	--	200	µA

CAN AC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
	Bit rate	Minimum 8 MHz clock	--	--	1	Mbit

Component Changes

This section lists the major changes in the component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
1.0.a	Updated rev number for release to the web.	No changes.
1.0.b	Updated DC and AC Electrical Characteristics section.	Information was incomplete.
	Minor datasheet edits.	Improve readability.

© Cypress Semiconductor Corporation, 2012. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC® is a registered trademark, and PSoC Creator™ and Programmable System-on-Chip™ are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.