THE SIMULATION PLATFORM FOR POWER ELECTRONIC SYSTEMS

PIL-FOC Demo for dspic33F MCUs Version 1.0

How to Contact Plexim:

☎	+41 44 533 51 00	Phone
	+41 44 533 51 01	Fax
✉	Plexim GmbH Technoparkstrasse 1 8005 Zurich Switzerland	Mail
@	`info@plexim.com`	Email
	`http://www.plexim.com`	Web

PIL-FOC Demo for dspic33F MCUs

Contents

Contents **iii**

Software Requirements 1

1 Getting Started **3**

Configuring the Hardware 4

Loading the Firmware 4

Configuring the PLECS Model 5

PIL Target 6

Testing the Communication 6

PIL Block 7

Running the PLECS Model 9

2 Processor-in-the-Loop **13**

Motivation 13

How PIL Works 14

PIL Modes 16

Configuring PLECS for PIL 17

Target Manager 17

Communication Links 18

PIL Block 20

3 PIL Framework **25**
Overview . . . 25
PIL Prep Tool . . . 26
Probes . . . 26
Read Probes . . . 26
Override Probes . . . 28
Calibrations . . . 31
Code Identity . . . 31
Remote Agent . . . 32
Communication Callbacks . . . 33
Serial Communication . . . 33
Parallel Communication . . . 33
Framework Integration and Execution . . . 34
Principal Framework Calls . . . 34
Control Callback . . . 38
Target Mode Switching . . . 39
Simulation Start and Termination . . . 40
Control Dispatching . . . 41
Task Synchronization at Start of Simulation . . . 43
Framework Configuration . . . 43
Configuration Constants . . . 44
Initialization Constants . . . 45

4 Microchip dsPIC33F Peripheral Models **47**
Introduction . . . 47
Microchip Motor Control PWM . . . 49
MCPWM Module Overview . . . 50
PWM Clock Control . . . 51
PWM Output Control and Resolution . . . 53
Special Event Trigger . . . 54
Interrupt Control . . . 55

Dead Time Generator 56
Summary of PLECS Implementation 57
Microchip Motor Control ADC 58
MCADC Module Overview 59
ADC Configuration 60
ADC Sampling and Conversion 62
Multi-channel ADC Sampling Mode 63
ADC Input Selection Mode 65
ADC Interrupt Logic 67
ADC Buffer Fill Mode 68
Summary of PLECS Implementation 68

5 Embedded Application **71**
Importing the MPLAB X Demo Project 71
Configuring the Project 71
Rebuilding the Project 72
Project Structure 72
Initialization and Task Dispatching 72
Control Law 73
Communication Interface 73
PIL Functionality 74

Before You Begin

This document contains instructions on how to test and evaluate the PLECS Processor-In-the-Loop (PIL) functionality in the context of a field-oriented motor control application.

Software Requirements

The demonstration is designed to be executed on a Windows machine (32-bit or 64-bit) with the following software installed:

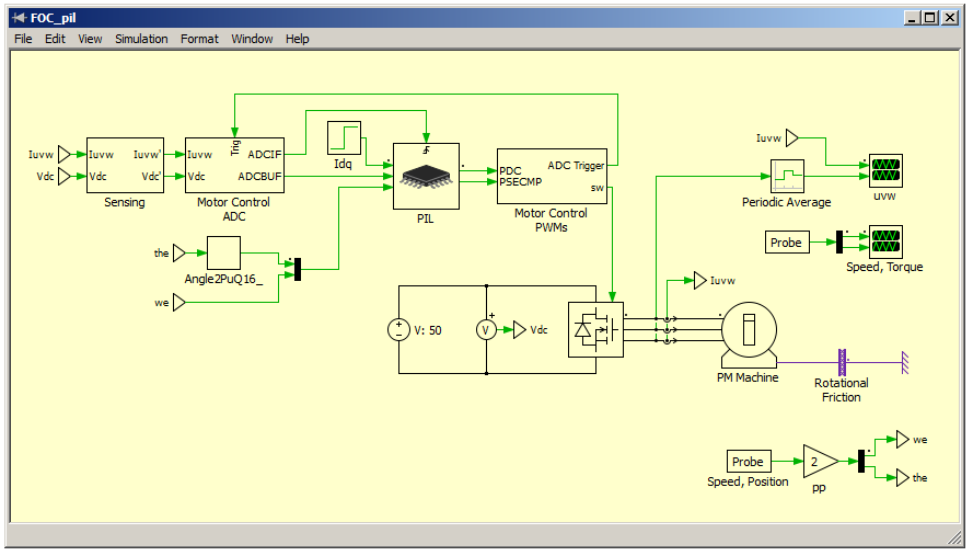
- PLECS Standalone or Blockset (version 3.7 or higher)
- MPLAB®X – Download from microchip.com.

A license is required to run PLECS and activate the PIL package. You can request such a license from Plexim at plexim.com. Copy the license file `license.dat` that will be supplied to you into the directory in which you have installed PLECS.

Getting Started

This chapter provides a hands-on demonstration of how control-code executing on a dsPIC33FJ128MC802 device can be tied into a PLECS simulation. More details about the Processor-in-the-Loop (PIL) concept and how embedded applications can be enabled for PIL is provided in the subsequent chapters.

The project is based on a basic Field Oriented Control (FOC) application, with the embedded code controlling the switches of a three-phase inverter powering a permanent magnet (PM) machine.



FOC demo model

The sample code is designed to execute on a Microstick-II evaluation board in

conjunction with a Microstick Plus peripheral panel.

Configuring the Hardware

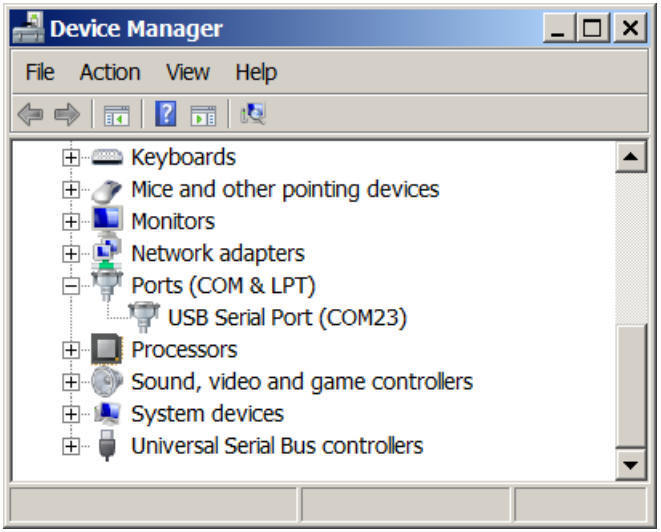
The slide-switch on the Microstick-II must be in the "A" position.

Loading the Firmware

Connect the PC with the evaluation system by means of two USB cables. First, make the connection to the Microstick-II board, then connect the Microstick Plus peripheral panel.

Open the Windows Device Manager and confirm the enumeration of a COM port.

You may have to install the MPC2200 drivers if the port is not enumerated.



COM port listed in device manager

The pre-compiled executable `j128mc.production.hex/elf` will be used to get us started. In MPLABX IPE (Integrated Programming Environment), select

the dsPIC33FJ128MC802 device and `j128mc.production.hex` firmware image. Then click **Connect** and **Program**.

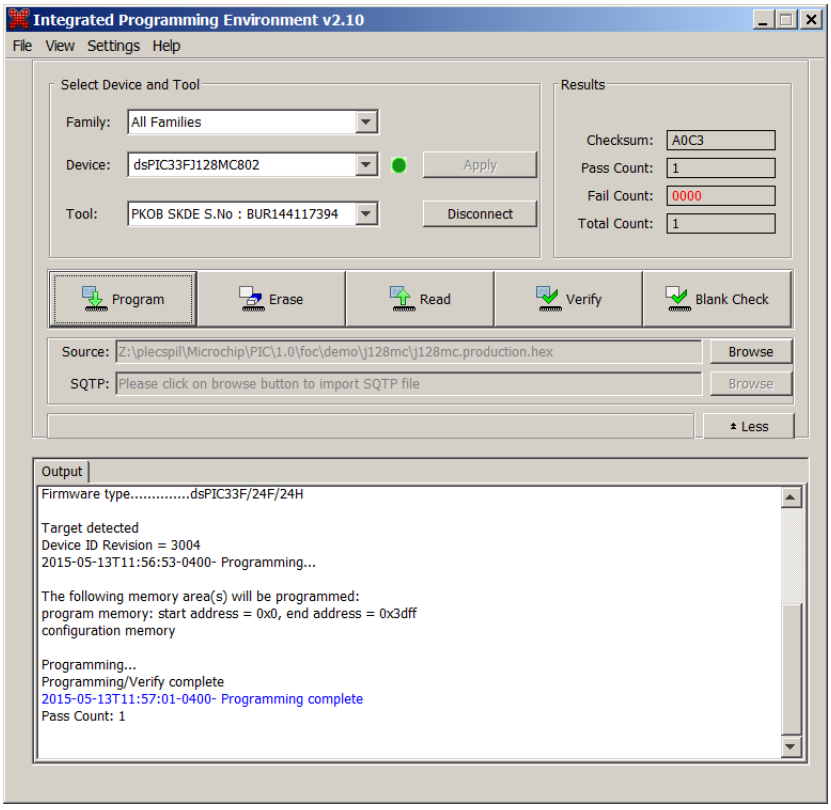


Figure 1.1: Flashing the dsPIC

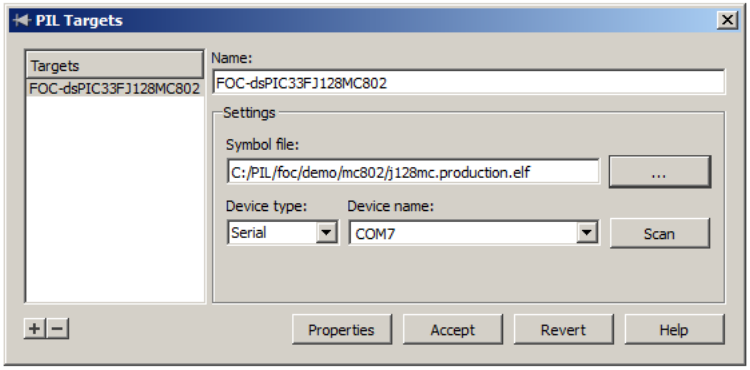
Once the reflashing completes, confirm that the red LED (D6) on the Microstick-II is blinking.

Configuring the PLECS Model

Start PLECS.

PIL Target

We now configure a PIL Target by means of the Target Manager. Open the target manager using the **Window** menu item **Target Manager**.



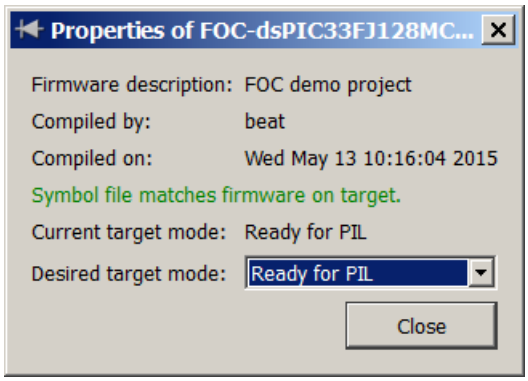
Target configuration

Click the **+** button and provide a name for the target. Next, select the **Symbol file** associated with the target by clicking the **...** button. The symbol file corresponds to the binary produced by the MPLAB codegen tools. Select the file `j128mc.production.elf`.

The remaining target configuration is the communication link. Select **Serial** from the **Device type** combo box. Then click on **Scan** and select the communication port that is being detected.

Testing the Communication

The target configuration can easily be verified by clicking the **Properties** button. This establishes communication with the target and displays diagnostics information in a new dialog window, as shown below.



Target properties

Confirm that the symbol file matches the firmware on the target. The **Target mode** should be **Ready for PIL**.

PIL Block

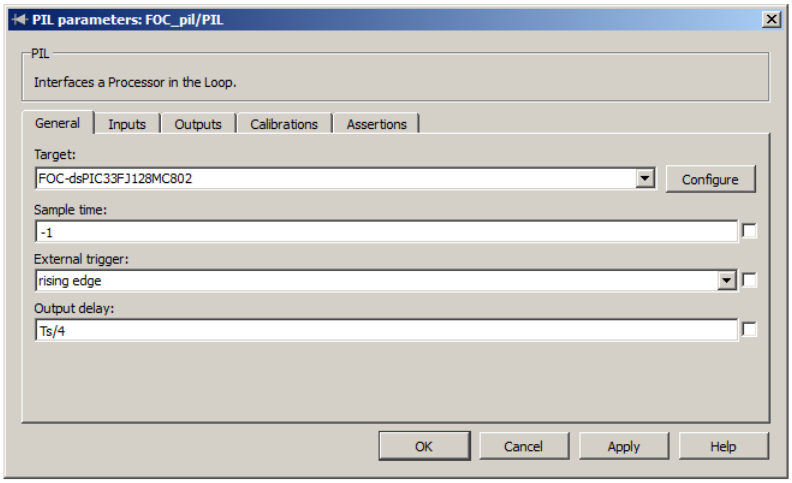
Now open the model named `FOC_pil` and double-click on the PIL block. Select the target that you defined in the target manager from the **Target** combo box.

Notice how the PIL block has been configured for an external trigger input. This allows the execution of the PIL block and associated embedded control code to be triggered by the ADC interrupt (ADCIF) event. The ADC, in turn, is triggered by an ePWM start-of-conversion (SOC) event in this example.

Activate the **Inputs** tab and see how the PIL block has been configured for the following three (3) inputs.

- `ControlVars.IdSet,IqSet` – Direct and quadrature current set-points (to be controlled by PI).
- `AdcOvrProbes.ADCRESULT0,1,2` – ADC conversion results (two currents and one voltage).
- `ControlVars.fluxPosition`, `ControlVars.we` – Position and speed of rotor.

The names of the signals listed above correspond to the variable names in the embedded code. As explained in subsequent chapters, a variable must be configured as an Override Probe to be used as a PIL block input. Notice how multiple Override Probes can be multiplexed into one input.



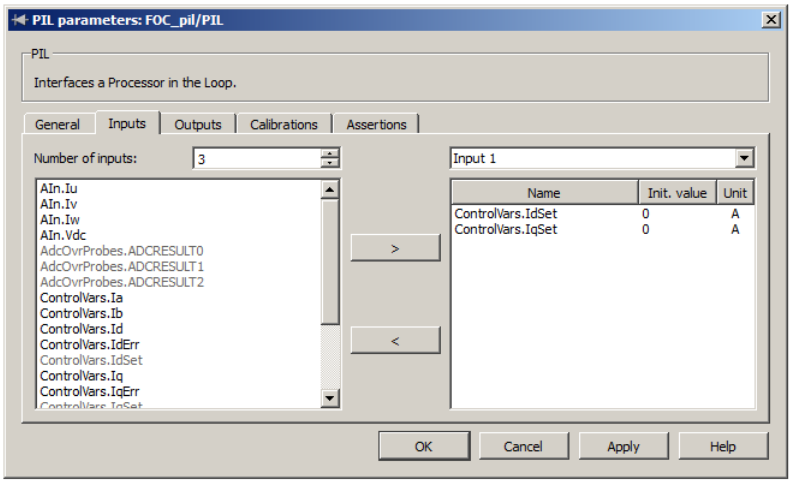
PIL block general configuration

The PIL block has been further configured for two (2) output (**Outputs** tab):

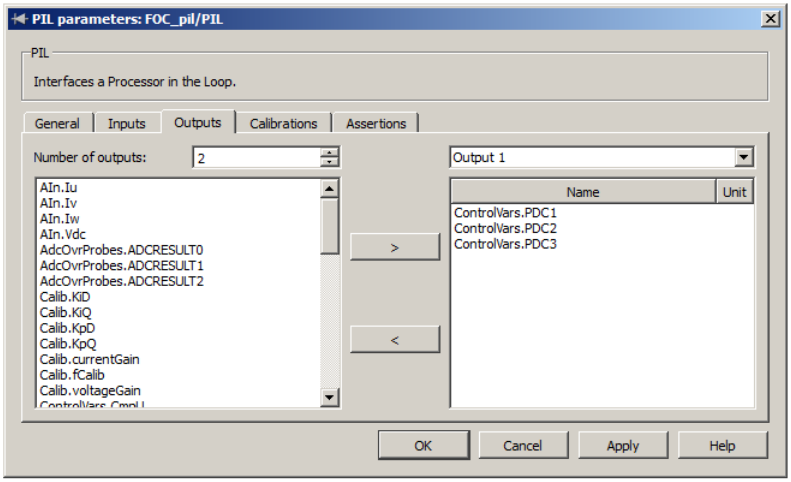
- `ControlVars.PDC1,2,3` – PWM peripheral duty cycle register values.
- `ControlVars.PSECMP` – PWM special event compare register value.

Again, the signal names correspond to the variable names in the embedded code. Variables must be configured as a Read Probe (or Override Probe) to be used as PIL block outputs. Notice how three Read Probes have been multiplexed into the same output.

Also accessible through the PIL block are the embedded code Calibrations. This tabs permits modifying and tuning settings in the embedded code, such as filter coefficients and regulator gains. Shown in the image below is an example in which the `Calib.KpQ` Calibration is modified from its default value.



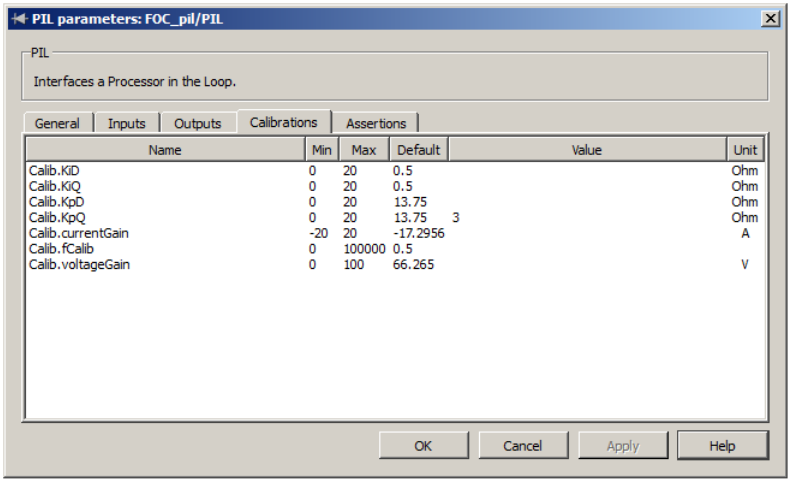
PIL block inputs



PIL block outputs

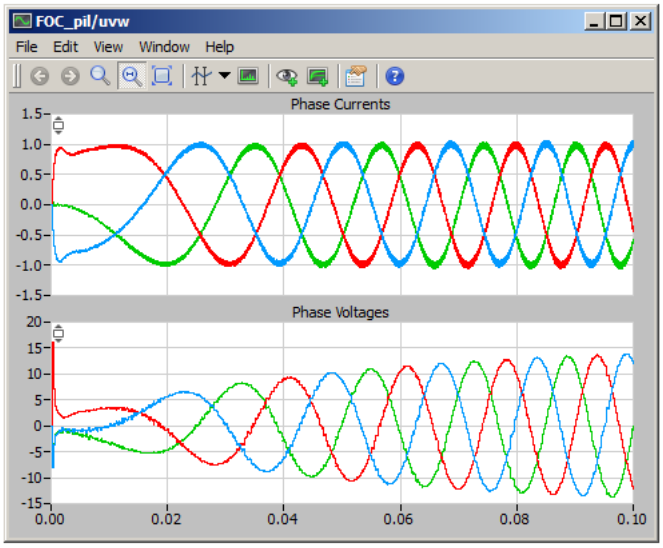
Running the PLECS Model

We can now run the simulation by pressing **Ctrl-T** or selecting **Start** from the **Simulation** menu.



PIL block calibrations

Observe how the embedded control algorithm is maintaining tight current regulation as the motor accelerates and the DC input voltage makes a step change.



PIL simulation result

2

Processor-in-the-Loop

As a separately licensed feature, PLECS offers support for *Processor-in-the-Loop* (PIL) simulations, allowing the execution of control code on external hardware tied into the virtual world of a PLECS model.

At the PLECS level, the PIL functionality consists of a specialized PIL block that can be found in the Processor-in-the-loop library, as well as the Target Manager, accessible from the **Window** menu. Also included with the PIL library are high-fidelity peripheral models of MCUs used for the control of power conversion systems.

On the embedded side, a *PIL Framework* library is provided to facilitate the integration of PIL functionality into your project.

Motivation

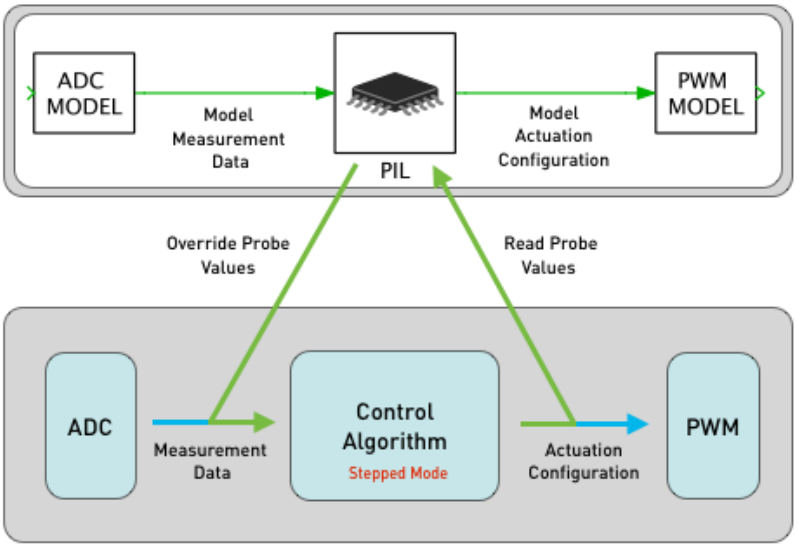
When developing embedded control algorithms, it is quite common to be testing such code, or portions thereof, by executing it inside a circuit simulator. Using PLECS, this can be easily achieved by means of a C-Script or DLL block. This approach is referred to as *Software-in-the-loop* (SIL). A SIL simulation compiles the embedded source code for the native environment of the simulation tool (e.g. Win64) and executes the algorithms within the simulation environment.

The PIL approach, on the other hand, executes the control algorithms on the real embedded hardware. Instead of reading the actual sensors of the power converter, values calculated by the simulation tool are used as inputs to the embedded algorithm. Similarly, outputs of the control algorithms executing on the processor are fed back into the simulation to drive the virtual environment. Note that SIL and PIL testing are also relevant when the embedded code is automatically generated from the simulation model.

One of the major advantages of PIL over SIL is that during PIL testing, actual compiled code is executed on the real MCU. This allows the detection of platform-specific software defects such as overflow conditions and casting errors. Furthermore, while PIL testing does not execute the control algorithms in true real-time, the control tasks *do* execute at the normal rate between two simulation steps. Therefore, PIL simulation can be used to detect and analyze potential problems related to the multi-threaded execution of control algorithms, including jitter and resource corruption. PIL testing can also provide useful metrics about processor utilization.

How PIL Works

At the most basic level, a PIL simulation can be summarized as follows:



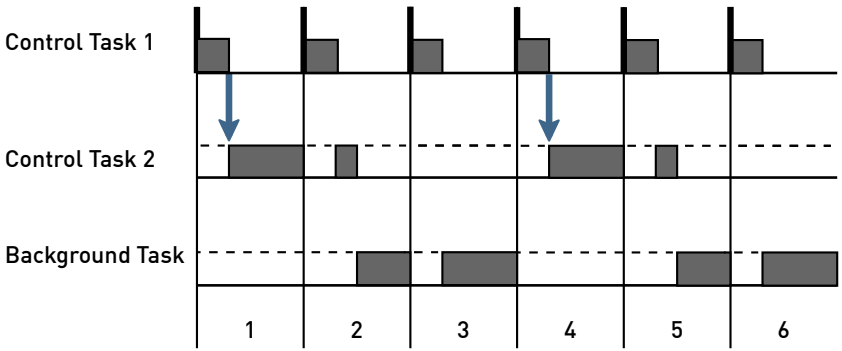
Principle of a PIL simulation

- Input variables on the target, such as current and voltage measurements, are overridden with values provided by the PLECS simulation.
- The control algorithms are executed for one control period.
- Output variables on the target, such as PWM peripheral register values, are read and fed back into the simulation.

We refer to variables on the target which are overridden by PLECS as *Override Probes*. Variables read by PLECS are called *Read Probes*.

While *Override Probes* are set and *Read Probes* are read the dispatching of the embedded control algorithms must be stopped. The controls must remain halted while PLECS is updating the simulated model. In other words, the control algorithm operates in a stepped mode during a PIL simulation. However, as mentioned above, when the control algorithms are executing, their behavior is identical to a true real-time operation. We therefore call this mode of operation *pseudo real-time*.

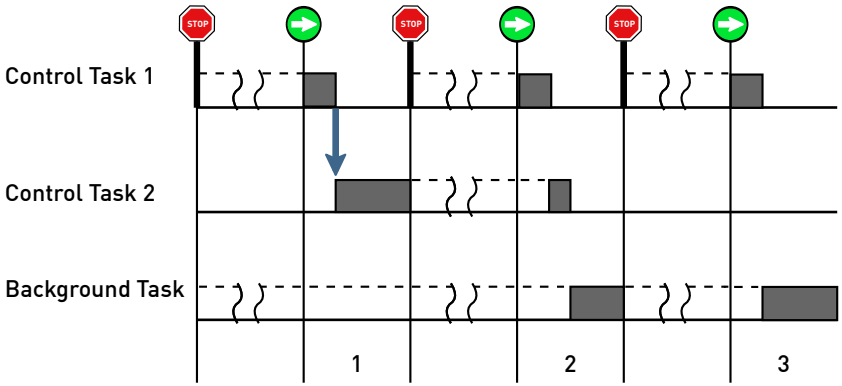
Let us further examine the pseudo real-time operation in the context of an embedded application utilizing nested control loops where fast high-priority tasks (such as current control) interrupt slower lower-priority tasks (such as voltage control). An example of such a configuration with two control tasks is illustrated in the figure below. With every hardware interrupt (bold vertical bar), the lower priority task is interrupted and the main interrupt service routine is executed. In addition, the lower priority task is periodically triggered using a software interrupt. Once both control tasks have completed, the system continues with the background task where lowest priority operations are processed. The timing in this figure corresponds to true real-time operation.



Nested Control Tasks

The next figure illustrates the timing of the same controller during a PIL simulation, with the *stop* and *go* symbols indicating when the dispatching of the control tasks is halted and resumed.

After the hardware interrupt is received, the system stops the control dispatching and enters a communication loop where the values of the Override Probes and Read Probes can be exchanged with the PLECS model. Once a new step request is received from the simulation, the task dispatching is



Pseudo real-time operation

restarted and the control tasks execute freely during the duration of one interrupt period. This pseudo real-time operation allows the user to analyze the control system in a simulation environment in a fashion that is behaviorally identical to a true real-time operation. Note that only the dispatching of the control tasks is stopped. The target itself is never halted as communication with PLECS must be maintained.

PIL Modes

The concept of using Override Probes and Read Probes allows tying actual control code executing on a real MCU into a PLECS simulation without the need to specifically recompile it for PIL.

You can think of Override Probes and Read Probes as the equivalent of test points which can be left in the embedded software as long as desired. Software modules with such test points can be tied into a PIL simulation at any time.

Often, Override Probes and Read Probes are configured to access the registers of MCU peripherals, such as analog-to-digital converters (ADCs) and pulse-width modulation (PWM) modules. Additionally, specific software modules, e.g. a filter block, can be equipped with Override Probes and Read Probes. This allows unit-testing the module in a PIL simulation isolated from the rest of the embedded code.

To permit safe and controlled transitions between real-time execution of the control code, driving an actual plant, and pseudo real-time execution, in con-

junction with a simulated plant, the following two PIL modes are distinguished:

- **Normal Operation** – Regular target operation in which PIL simulations are inhibited.
- **Ready for PIL** – Target is ready for a PIL simulation, which corresponds to a safe state with the power-stage disabled.

The transition between the two modes can either be controlled by the embedded application, for example based on a set of digital inputs, or from PLECS using the Target Manager.

Configuring PLECS for PIL

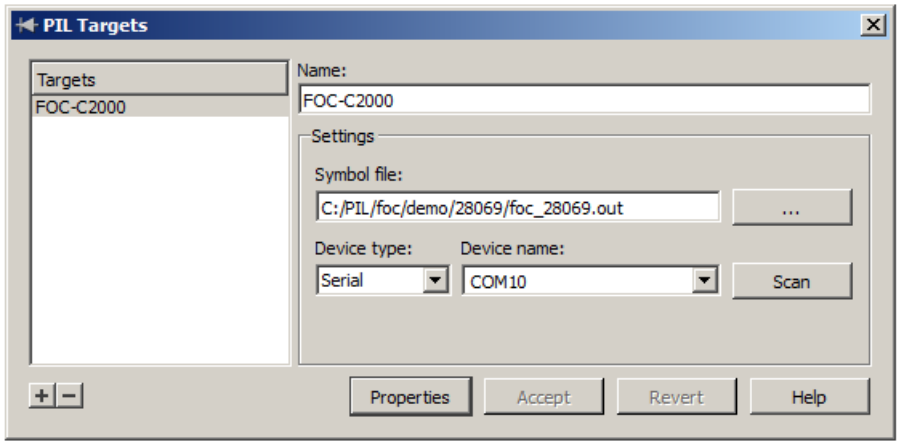
Once an embedded application is equipped with the PIL framework, and appropriate Override Probes and Read Probes are defined, it is ready for PIL simulations with PLECS.

PLECS uses the concept of *Target Configurations* to define global high-level settings that can be accessed by any PLECS model. At the circuit level, the *PIL block* is utilized to define lower level configurations such as the selection of Override Probes and Read Probes used during simulation.

This is explained in further detail in the following sections.

Target Manager

The high-level configurations are made in the *Target Manager*, which is accessible in PLECS by means of the corresponding item in the **Window** menu. The target manager allows defining and configuring targets for PIL simulation, by associating them with a symbol file and specifying the communication parameters. Target configurations are stored globally at the PLECS level and are not saved in `*.plecs` or Simulink files. An example target configuration is shown in the figure below.



Target Manager

The left hand side of the dialog window shows a list of targets that are currently configured. To add a new target configuration, click the button marked **+** below the list. To remove the currently selected target, click the button marked **-**. You can reorder the targets by clicking and dragging an entry up and down in the list.

The right hand side of the dialog window shows the parameter settings of the currently selected target. Each target configuration must have a unique **Name**.

The target configuration specifies the **Symbol file** and the communication link settings.

The symbol file is the binary file (also called “object file”) corresponding to the code executing on the target. PLECS will obtain most settings for PIL simulations, as well as the list of Override Probes and Read Probes and their attributes, from the symbol file.

Communication Links

A number of links are supported for communicating with the target. The desired link can be selected in the **Device type** combo box. For communication links that allow detecting connected devices, pressing the **Scan** button will populate the **Device name** combo box with the names of all available devices.

Serial Device

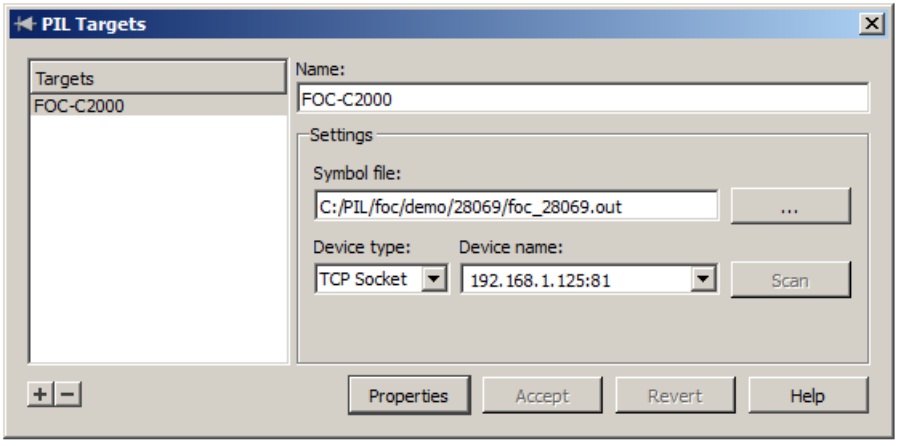
The **Serial device** selection corresponds to conventional physical or virtual serial communication ports. On a Windows machine, such ports are labeled `COMn`, where `n` is the number of the port.

FTDI Device

If the serial adapter is based on an FTDI chip, the low-level FTDI driver can be used directly by selecting the **FTD2XX** option. This device type offers improved communication speed over the virtual communication port (VCP) associated with the FTDI adapter.

TCP/IP Socket

The communication can also be routed over a TCP/IP socket by selecting the **TCP Socket** device type.



TCP/IP Communication

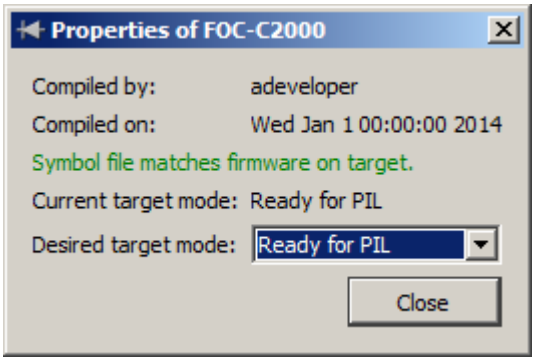
In this case the **Device name** corresponds to the IP address (or URL) and port number, separated by a colon (:).

TCP/IP Bridge

The **TCP Bridge** device type provides a generic interface for utilizing custom communication links. This option permits communication over an external application which serves as a “bridge” between a serial TCP/IP socket and a custom link/protocol.

Target Properties

By pressing the **Properties** button, target information can be displayed as shown in the figure below.

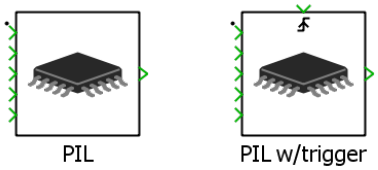


Target Properties

In addition to reading and displaying information from the symbol file, PLECS will also query the target for its identity and check the value against the one stored in the symbol file. This verifies the device settings and ensures that the correct binary file has been selected. Further, the user can request for a target mode change to configure the embedded code to run in **Normal Operation** mode or in **Ready for PIL** mode.

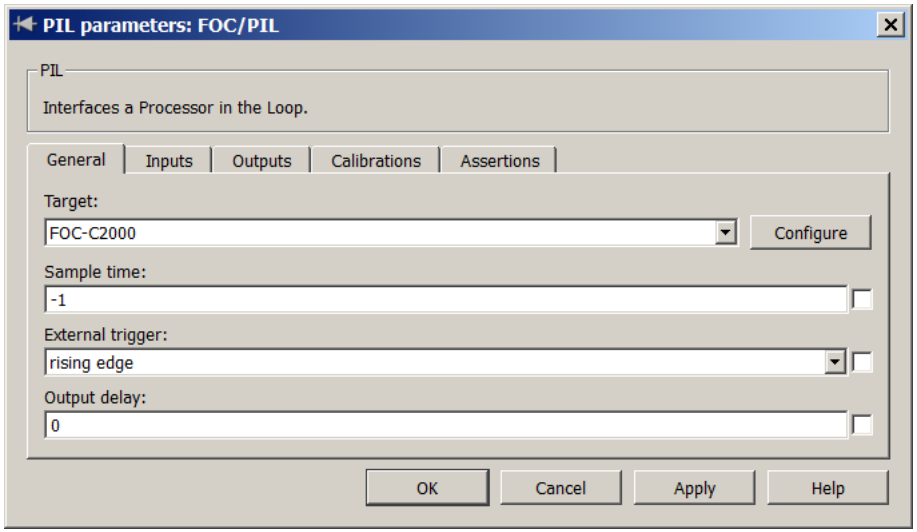
PIL Block

The PIL block ties a processor into a PLECS simulation by making Override Probes and Read Probes, configured on the target, available as input and output ports, respectively.



PIL Block

A PIL block is associated with a target defined in the target manager, which is selected from the **Target** combo box. The **Configure...** button provides a convenient shortcut to the target manager for configuring existing and new targets.



PIL Block General Tab

The execution of the PIL block can be triggered at a fixed **Discrete-Periodic** rate by configuring the **Sample time** to a positive value. As with other PLECS components, an **Inherited** sample time can be selected by setting the parameter to **-1** or **[-1 0]**.

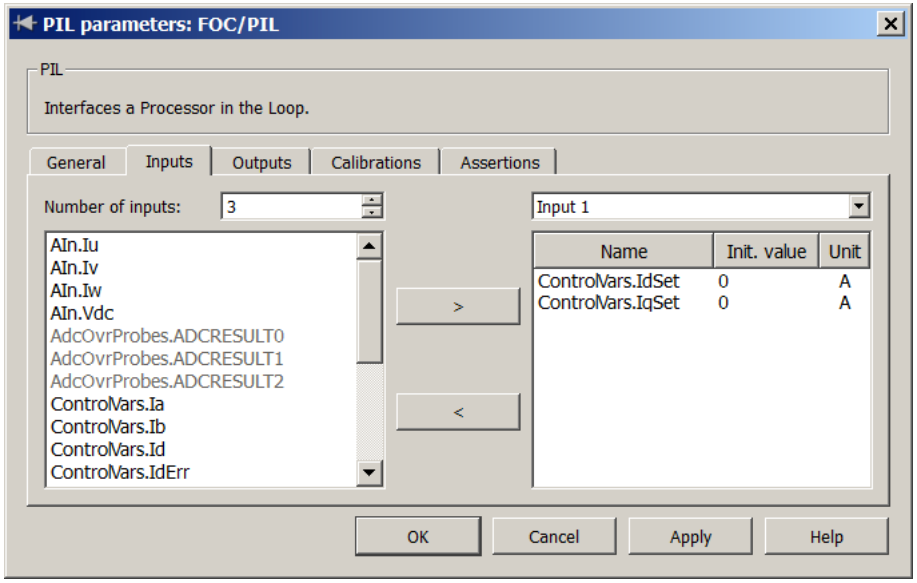
A trigger port can be enabled using the **External trigger** combo box. This is useful if the control interrupt source is part of the PLECS circuit, such as an ADC or PWM peripheral model.

Typically, an **Inherited** sample time is used in combination with a trigger port. If a **Discrete-Periodic** rate is specified, the trigger port will be sampled at the specified rate.

Similar to the DLL block, the **Output delay** setting permits delaying the output of each simulation step to approximate processor calculation time.

Note Make sure the value for the **Output delay** does not exceed the sample time of the block, or the outputs will never be updated.

A delay of **0** is a valid setting, but it will create direct-feedthrough between inputs and outputs.



PIL Block Inputs Tab

The PIL block extracts the names of Override Probes and Read Probes from the symbol file selected in the target configuration and presents lists for selection as input and output signals, as shown in the figure above.

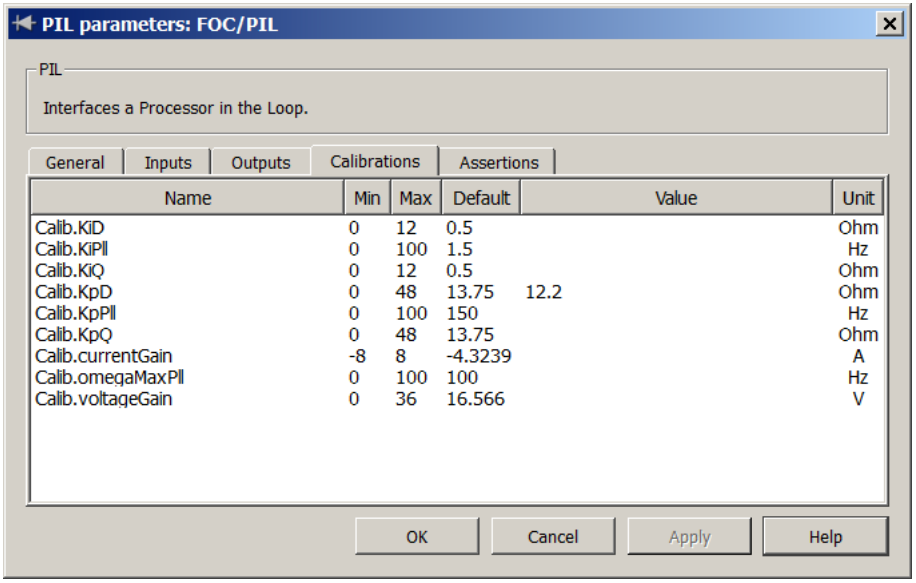
The number of inputs and outputs of a PIL block is configurable with the **Number of inputs** and **Number of outputs** settings. To associate Over-

ride Probes or Read Probes with a given input or output, select an input/output from the combo box on the right half of the dialog. Then drag the desired Override Probes or Read Probes from the left into the area below or add them by selecting them and clicking the **>** button. To remove an Override Probe or Read Probe, select it and either press the **Delete** key or **<** button.

Note It is possible to multiplex several Override/Read Probe signals into one input/output. The sequence can be reordered by dragging the signals up and down the list.

Starting with PLECS 3.7, the PIL block allows setting initial conditions for Override Probes.

Also new with PLECS 3.7 is the Calibrations tab, which permits modifying embedded code settings such as regulator gains and filter coefficients.



PIL Block Calibrations Tab

Calibrations can be set in the **Value** column. If no entry is provided, the embedded code will use the default value as indicated in the **Default** column.

3

PIL Framework

Plexim provides and maintains *PIL Frameworks* for specific processor families, which encapsulate all the necessary embedded functionality for PIL operation. Using the PIL framework, your C or C++ based embedded applications can be enabled for PIL with minimal effort.

Currently, such frameworks and associated demo applications are available for the Texas Instruments (TI) C2000™, ST Microelectronics 32bit F4 and the Microchip dsPIC33F MCU families. However, support for other platforms can be developed, as long as the following basic requirements are met:

- The code generation tools (compiler and linker) must be able to generate binary files of the ELF format containing DWARF debugging information.
- The address width of the processor cannot exceed 32 bit.
- The least addressable unit (LAU) of the processor must be no larger than 16-bit.

Overview

The fundamental operation of a PIL simulation consists of overriding and reading variables in the embedded application, and synchronizing the execution of the control task(s) with the simulation of a PLECS model. The PIL framework therefore provides the following functionality:

- Read Probes for reading the values of variables in the embedded code executing on the target and feeding the information into the simulation model.
- Override Probes for overriding variables in the embedded code with values obtained from the simulation.
- A method to uniquely identify the software executing on the target.
- A remote agent, capable of communicating with PLECS and interpreting commands related to PIL operation.

- A mechanism for stopping and starting the execution of the control tasks.
- A means to provide configuration parameters to PLECS, such as the communication baudrate.

Starting with PLECS 3.7, the PIL framework also supports *Calibrations*, which are embedded–code parameters such as filter coefficients and regulator gains. Calibrations can be modified in the PLECS environment during the initialization of a PIL simulation and allow running multiple simulations with different settings without the need for recompiling the embedded code (e.g. for the tuning of regulators).

PIL Prep Tool

To facilitate defining and configuring PIL probes and calibrations, starting with PLECS 3.7, a *PIL Prep Tool* utility is provided as part of the PIL framework.

The PIL Prep Tool parses the embedded code for PIL specific macros, and automatically generates auxiliary files to be compiled and linked with the embedded code. These auxiliary files contain functions for initializing probes and calibrations, as well as special symbols which describe to PLECS the scaling and formatting of the probes/calibrations. The generated files further include a globally unique identifier (GUID) allowing PLECS to identify the embedded code.

The PIL Prep Tool must be called as a pre-build step. Its integration into an embedded project is specific to the compiler and integrated development environment (IDE) used. Please refer to the PIL demo projects for more information.

Probes

Read Probes

Read Probes are variables in the embedded code which are configured for read access by PLECS. Any global variable can be configured as a Read Probe by means of the `PIL_READ_PROBE` macro. For example, the statement below defines and configures variable `Vdc` for read access by PLECS.

```
1      PIL_READ_PROBE(uint16_t , Vdc, 10, 5.0, "V");
```

The PIL_READ_PROBE macro results in a simple variable definition, e.g. uint16_t Vdc, but is also recognized by the PIL Prep Tool, which places the following statement in the auto generated file:

```
1      PIL_SYMBOL_DEF(Vdc, 10, 5.0, "V");
```

The PIL_SYMBOL_DEF macro expands into the definition of a specially formatted and statically initialized helper structure of type const.

```
1      typedef struct
2      {
3          int q;          //!< fixed-point location
4          float ref;      //!< reference value
5          char *unit;     //!< unit string
6      } pil_var;
7
8      const pil_var PIL_V_Vdc = {10, 5.0, "V"}
```

PLECS searches for PIL_V symbols when parsing the binary file selected in the target manager, and uses the information of the PIL_V symbols to translate between the raw values stored in the Read Probe and the corresponding physical value to be used in the simulation.

In the above example, the global variable Vdc is configured as a Q10 with a reference of 5V. Hence, an integer value of 512 in this variable will be converted by PLECS to $\frac{512}{2^{10}} * 5\text{V} = 2.5\text{V}$.

A fixed point variable can be configured as a unitless number by using a reference value of 1.0 and setting an empty string ("") for the unit.

The same approach can be used to configure floating point variables as Read Probes.

```
1      PIL_READ_PROBE(float, MotorSpeed, 0, 1.0, "rpm");
```

The third parameter of the PIL_READ_PROBE macro, i.e. the fixed point location, is ignored with probed floating point variables. However, it is possible to specify reference values for floating point variables. For example, the macro below configures MotorSpeed with a reference of 1800 rpm. Hence, a value of 0.5 in this variable will be converted to $0.5 * 1800\text{rpm} = 900\text{rpm}$.

It is also possible to configure structure members, as shown below.

```
1      struct BATTERY {
2          PIL_READ_PROBE(int16_t, voltage, 10, 5.0, "V");
3      };
```

Override Probes

Override Probes, i.e. variables in the embedded code that can be overridden by PLECS, are defined with the `PIL_OVERRIDE_PROBE` macro as illustrated below.

```
1      struct BATTERY {
2          PIL_OVERRIDE_PROBE(int16_t, voltage, 10, 5.0, "V");
3      };
4
5      struct BATTERY MyBattery;
```

The `PIL_OVERRIDE_PROBE` macro expands into a variable definition that is augmented by two helper symbols which permit the `MyBattery.voltage` variable to be overridden by PLECS.

```
1      struct BATTERY {
2          int16_t voltage;
3          int16_t voltage_probeV;
4          int16_t voltage_probeF;
5      };
```

While parsing a binary file for symbol information, PLECS detects variables with matching `_probeF` and `_probeV` definitions and identifies those as Override Probes.

In addition, the PIL Prep Tool will recognize the `PIL_OVERRIDE_PROBE` macro and generate the following auxiliary macro as described in the Read Probe section:

```
1      PIL_SYMBOL_DEF(MyBattery_voltage, 10, 5.0, "V");
```

Note Only variables defined as Override Probes are configurable as inputs for the PIL block.

An Override Probe is similar to a toggle switch with the following two states:

- **Feedthrough** – The Override Probe value is provided by the embedded application
- **Override** – The Override Probe value is provided by PLECS

The state of an Override Probe can be switched dynamically at runtime and is stored in the `_probeF` helper variable.

With this approach, the same build of the embedded application can be used to control actual hardware or be tested in a PIL simulation, by simply switching the mode of Override Probes, without recompiling.

To properly interact with PLECS, the embedded code must access the Override Probes exclusively by the following set of macros:

Override Probe Macros

Macro	Description
`INIT_OPROBE(probe)`	Initializes an Override Probe. Must be called during the initialization of the embedded program.
`SET_OPROBE(probe, value)`	Assigns a value to an Override Probe.

The PIL Prep Tool will generate a function called `PilInitOverrideProbes()` which contains `INIT_OPROBE` calls for all Override Probes. This function must be called during the initialization phase of the embedded code before any Override Probes are used.

If an Override Probe is in the feedthrough state, the **value** assigned to the macro is written into **probe**. Otherwise, the override value supplied by PLECS is used, which is stored in the `_probeV` helper variable.

An example for adding Override Probes to existing code is given in the following two listings.

Original code without use of Override Probes

```
1       Battery.voltage = measureBattVolt();
2
3       PLX_VECT_parkRot(ControlVars.Ia, ControlVars.Ib, \
4           &ControlVars.Id, &ControlVars.Iq, \
5           ControlVars.fluxPosSin, ControlVars.fluxPosCos);
```

Assume that during PIL simulations, we would like to override the variable `Battery.voltage` as well as the values of `ControlVars.Id` and `ControlVars.Iq`. While the battery voltage is updated by a simple write access, the Id and Iq variables are modified by the `PLX_VECT_parkRot(...)` function via pointers, which need special handling for the `SET_OPROBE` macro integration.

The next listing illustrates how `SET_OPROBE` is properly used in this example.

Use of Override Probes

```
1       SET_OPROBE(Battery.voltage, measureBattVolt());
2
3       int16_t id, iq;
4
5       PLX_VECT_parkRot(ControlVars.Ia, ControlVars.Ib, \
6           &id, &iq, \
7           ControlVars.fluxPosSin, ControlVars.fluxPosCos);
8
9       SET_OPROBE(ControlVars.Id, id);
10      SET_OPROBE(ControlVars.Iq, iq);
```

For the battery voltage, the assignment can simply be replaced by the SET_OPROBE macro. For the Id and Iq values, auxiliary variables are used, updated by the `PLX_VECT_parkRot(...)` function, and subsequently assigned to the Override Probes.

Note The `SET_OPROBE` macro must be used whenever a value is assigned to an Override Probe. A direct assignment using the equal (=) statement will result in unpredictable behavior.

Calibrations

Calibrations are variables used to configure algorithms in the embedded code, such as filter coefficients, thresholds, timeouts and regulator gains.

The PIL framework provides the `PIL_CALIBRATION` macro for a convenient definition of such calibrations. For example, the statement below declares and configures variable `Kp` as a PIL calibration.

```
1      PIL_CALIBRATION(int16_t, Kp, 10, 5.0, "Ohm", 0, 10.0, 0.5);
```

The first five parameters of the `PIL_CALIBRATION` macro are identical to the definition of a Read Probe. Accordingly, the macro expands into a simple variable definition `uint16_t Kp`.

The additional three parameters define the allowable range of values for the Calibration as well as its default value.

In the above example, the allowable range for Kp is $0 - 10\Omega$. Upon initialization, Kp is set to 0.5Ω.

The `PIL_CALIBRATION` macro is interpreted by the PIL Prep Tool to generate a `PIL_SYMBOL_CAL_DEF` macro. Similar to `PIL_SYMBOL_DEF`, this macro produces the necessary information for PLECS to properly interpret and handle the calibration. The PIL Prep Tool also generates a function called `PilInitCalibrations()` which sets all Calibrations to default values. This function must be called during the initialization phase of the embedded code before any calibrations are used. It is also important that this function be called in the `PIL_CLBK_TERMINATE_SIMULATION` callback to revert changes made during a PIL simulation.

Code Identity

PLECS accesses Override Probes, Read Probes and Calibrations by address (as opposed to name). The PIL block extracts the address of a given variable from the debugging information contained in the binary file supplied to the Target Manager. It is therefore important to ensure the selected binary file matches the code that is actually executing on the target, or erroneous memory locations will be accessed. This is achieved by comparing a globally unique

identifier (GUID) stored in the binary file with the value reported by the target. PLECS performs this check at the beginning of a simulation, as well as when the PIL block is opened. As explained in section "Target Manager" (on page 17), the target manager can be used to verify the match of the selected binary file.

The GUID is generated at compile time by the PIL Prep Tool. Additionally, macros for the compile time, and log-on name of the person who compiled the code are created.

```
1      #define CODE_GUID {0xA8,0x45,0x11,0xDE,0x05,0x4C,0xAC,0x41}
2      #define COMPILE_TIME_DATE_STR "Sun May 30 10:11:43 2010"
3      #define USER_NAME "john doe"
```

The value of `CODE_GUID` is passed to the PIL framework during initialization; see "Framework Configuration" (on page 43). The value must also be assigned to the `PIL_D_Guid` constant as follows:

```
1      PIL_CONST_DEF(unsigned char, Guid[], CODE_GUID);
```

The other two macros can be used for diagnostics purposes using PIL constants, as demonstrated in section "Configuration Constants" (on page 44).

Remote Agent

The *remote agent* services the communication link with PLECS and processes commands received from PLECS to access Override Probes and Read Probes, and to step the control code during a PIL simulation.

The remote agent supports both parallel and serial communications, but is agnostic of the hardware specific details of the communication link.

The user of the PIL framework is responsible for implementing the driver for a specific communication link, i.e. for configuration of hardware and basic reception and transmission of data.

Communication Callbacks

The PIL framework interacts with the application specific communication driver by *communication callback functions*. Two callbacks exist:

- `CommCallback()` – Called at each system interrupt from `PIL_beginInterruptCall()`.
- `BackgroundCommCallback()` – Periodically called from `PIL_backgroundCall()`.

A given communication link might use either or both callbacks for its implementation. For implementing serial or parallel data exchange with the framework, the user needs to utilize the input and output functions presented in the following sections. The callback functions are registered with the framework as described on page 43.

Serial Communication

For serial communication, the remote agent utilizes a simple network layer with message framing and error checking, making the protocol suitable for a wide range of links such as RS-232, RS-485, TCP/IP and CAN.

To ensure no characters are dropped during a serial communication, the `CommCallback()` from the interrupt should be used to service the link.

A typical implementation of a serial communication callback is shown in the SCI callback listing.

Notice the use of the following two functions:

- `PIL_RA_serialIn(...)` – For the reception of characters.
- `PIL_RA_serialOut(...)` – For the transmission of characters.

Parallel Communication

For parallel communication, complete messages are directly exchanged with the framework as 16-bit integer arrays. The parallel link does not utilize any framing or checksum. This link is therefore suited for exchanging messages via shared memory where risk of transmission errors is negligible.

Parallel communications are typically serviced by the callback made from the background loop.

- `PIL_RA_parallelIn(...)` – For the reception of a message.
- `PIL_RA_parallelOut(...)` – For the transmission of a message.

SCI callback

```
1       void SCIPoll()
2       {
3           while(SciaRegs.SCIFFRX.bit.RXFFST != 0)
4           {
5               // a character has been received
6               PIL_RA_serialIn((int16)SciaRegs.SCIRXBUF.all);
7           }
8
9           int16_t ch;
10          if(SciaRegs.SCICTL2.bit.TXRDY == 1)
11          {
12              // link is ready for transmission
13              if(PIL_RA_serialOut(&ch))
14              {
15                  SciaRegs.SCITXBUF = ch;
16              }
17          }
18      }
```

Framework Integration and Execution

Principal Framework Calls

The PIL framework provides the following two principal functions which must be called periodically by the embedded application to enable PIL functionality:

- `PIL_beginInterruptCall()` – Framework call from interrupt.
- `PIL_backgroundCall(...)` – Framework call from background loop.

The `PIL_beginInterruptCall()` must be added at the beginning of the main interrupt service routine, while the `PIL_backgroundCall(...)` is called periodically from the background task.

The actions performed by those calls depends on whether a PIL simulation is running or not.

In the following, the concept of the PIL integration is further explained for a system with nested control tasks (see code snippet below).

In this example, the first control task is triggered by a hardware interrupt related to the system counter. A divider is used to dispatch a second, lower priority task. When the divider reaches a specified value, the second control task is dispatched by a software interrupt.

Control Task Dispatching

```
1       /**
2        * Main interrupt routine
3        */
4       Void TickFxn(UArg arg)
5       {
6           PIL_beginInterruptCall();
7
8           // fast control task
9           ControlTask1();
10
11          // slow control task
12          divider++;
13          if(divider == TASK2_PERIOD)
14          {
15              divider = 0;
16              Swi_post(Swi);
17          }
18      }
19
20      /**
21       * Software interrupt for slow control task
22       */
23      Void SwiFxn(UArg arg0, UArg arg1)
24      {
25          ControlTask2();
26      }
27
28      /**
29       * Background task
30       */
31      Void BackgroundTaskFxn(Void)
32      {
33          PIL_backgroundCall();
34      }
```

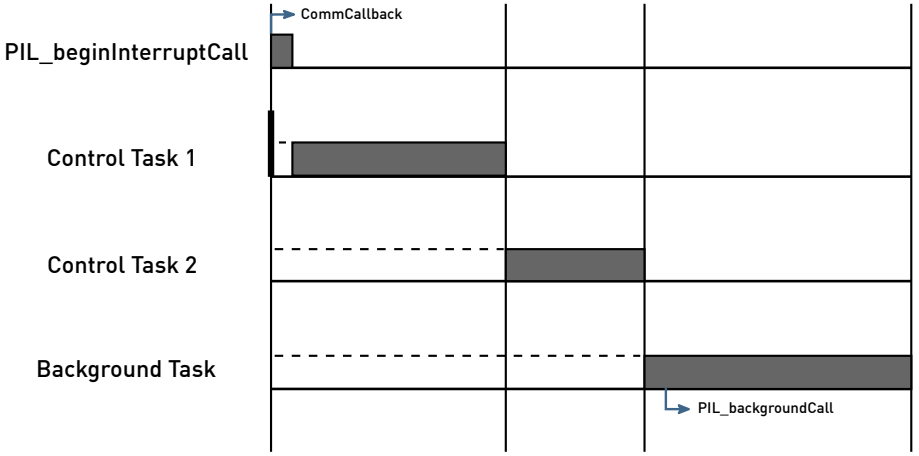
	Real-time	Pseudo Real-time
PIL_beginInterruptCall	CommCallback	CommCallback BackgroundCommClbk Message Evaluation PIL Cmd Handling
PIL_backgroundCall	BackgroundCommClbk Message Evaluation PIL Cmd Handling	N/A

Mode-specific actions during framework execution

Assuming the slow task takes longer than a hardware interrupt period, the second control task is interrupted several times before its execution is finished.

Now let us examine the operation of the framework in both real-time and pseudo real-time mode.

The figure on page 36 shows the framework operation in non-PIL (real-time) mode.



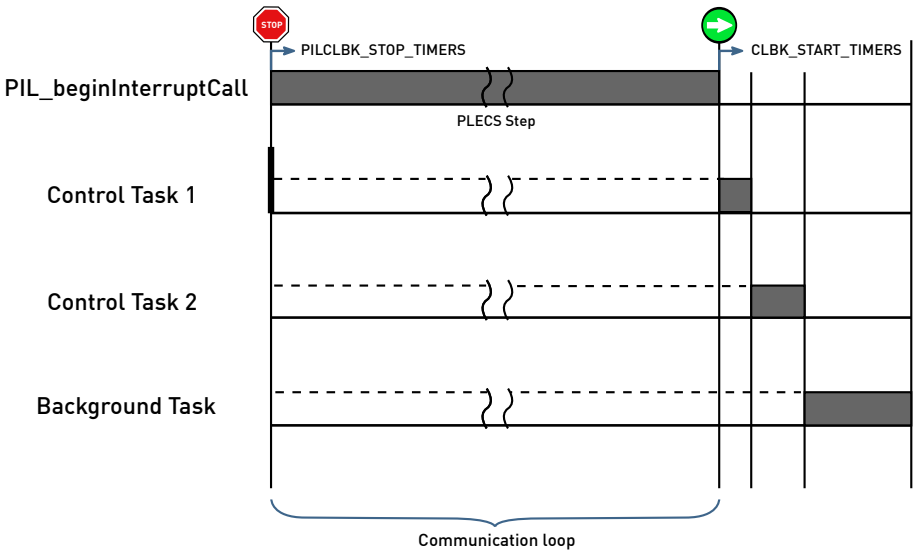
PIL framework during real-time operation

At the beginning of the hardware interrupt service routine, the `PIL_beginInterruptCall()` is executed, which, in real-time mode, only calls

the registered `CommCallback` function. As already mentioned, this callback should be used to service the link for a serial communication to ensure no characters are dropped.

Note During real-time operation, the PIL framework must have a minimal influence on the timing of the dispatched control tasks. Therefore the `CommCallback` function must be implemented as efficiently as possible.

As its name suggests, `PIL_backgroundCall(...)` function is executed from the background loop, which in turn calls the `BackgroundCommCallback()`, if configured. The `PIL_backgroundCall(...)` also parses incoming messages that are buffered by the communication callback functions, and processes PIL commands.



PIL framework during pseudo real-time operation

The next figure shows the system behavior during a PIL simulation, i.e. in pseudo real-time mode, where control task execution is paced and synchronized with the simulation of a PLECS model.

At the start of the hardware interrupt service routine, the task dispatching stops and the system enters a communication loop.

In this loop, both communication callbacks and the command parsing functions are executed. This is different from true real-time mode, where the background communication callback and the command parsing functions are called from the background loop.

Once a request for a new control step is received, the framework resumes the control task dispatching and continues in free mode until the next hardware interrupt occurs. Note that in pseudo real-time operation, the `PIL_backgroundCall()` has no effect.

Control Callback

The transition between different operating modes as well as the pseudo real-time operation require application-specific actions, implemented by means of a *Control Callback*.

For example, when entering the Ready for PIL mode, the power actuation must be turned off, e.g. by disabling the PWM outputs. Also, during a PIL simulation the peripherals providing the timing to the control algorithms must be stopped and restarted, as indicated by the arrows labeled PIL_CLBK_STOP_TIMERS and PIL_CLBK_START_TIMERS.

These control actions are provided by a single callback function registered during the framework initialization, and subsequently executed with an argument specifying the specific action to be taken.

Consequently, the implementation of this callback typically consists of a `switch` statement as shown below:

The following control-callback actions are defined and called during the framework execution:

- `PIL_CLBK_ENTER_NORMAL_OPERATION_REQ` – Called when the target mode "Normal Operation" has been requested. The application must indicate that it has entered normal operation by executing `PIL_inhibitPilSimulation()`.
- `PIL_CLBK_LEAVE_NORMAL_OPERATION_REQ` – Called when the target mode "Ready for PIL" has been requested. The application must confirm that it is ready for PIL simulations by executing `PIL_allowPilSimulation()`.
- `PIL_CLBK_PREINIT_SIMULATION` – Called before transitioning to a PIL simulation. Can be used to reconfigure task dispatching, for example if an MCU coprocessor such as the TI CLA is to be tied into the PIL loop. Interrupts are disabled when this call is made.

```
1  void PilCallback(PIL_CtrlCallbackReq_t aCallbackReq)
2  {
3      switch(aCallbackReq)
4      {
5          case PIL_CLBK_STOP_TIMERS:
6              //application specific code
7              break;
8          case PIL_CLBK_START_TIMERS:
9              //application specific code
10             break;
11             .
12             .
13             .
14         default:
15             //catching an undefined callback
16             break;
17     }
18 }
```

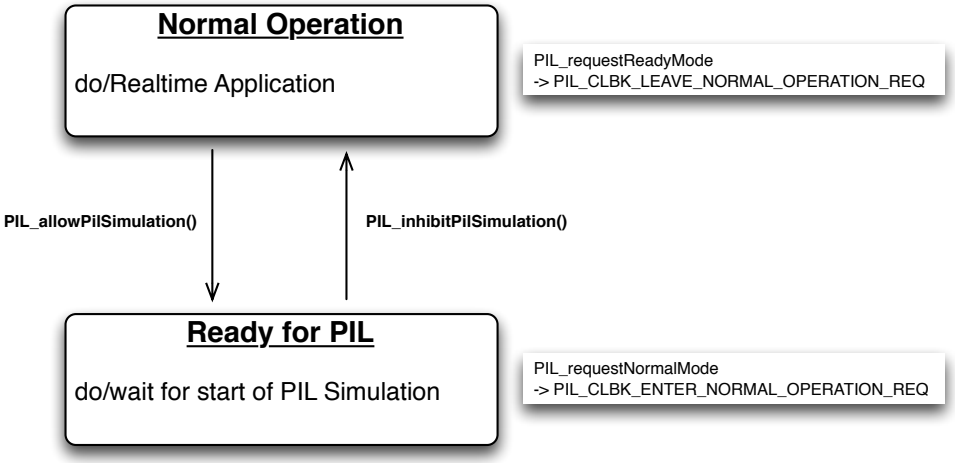
- `PIL_CLBK_INITIALIZE_SIMULATION` – Called at the beginning of a PIL simulation. Used to reset the controller(s) and control task dispatching to initial conditions.
- `PIL_CLBK_TERMINATE_SIMULATION` – Called at the end of a PIL simulation.
- `PIL_CLBK_STOP_TIMERS` – Called at the beginning of the control interrupt when in PIL mode (pseudo real-time operation). Used to stop all timers and counters related to the control tasks.
- `PIL_CLBK_START_TIMERS` – Called immediately before resuming the control task(s) when in PIL mode (pseudo real-time operation). Used to restart all timers and counters related to the control tasks.

In the following sections, the different actions are further described in context of when they are called during the operation of the PIL framework. Please also review the example projects provided by Plexim for further details and control callback implementation examples.

Target Mode Switching

As described in the section “PIL Modes” (on page 16) the PIL framework distinguishes between the two target modes.

In Normal Operation mode, the target executes in true real-time operation driving the load with an active power stage. PIL simulations are inhibited



PIL target modes and mode change requests

in this mode due to the power stage being active. A PIL simulation can only be started if the target is in Ready for PIL mode, which corresponds to a safe state in which the power stage is disabled. As explained in the prior section, the code for enabling or disabling the power stage is application specific and must be provided by the user via the corresponding control callback.

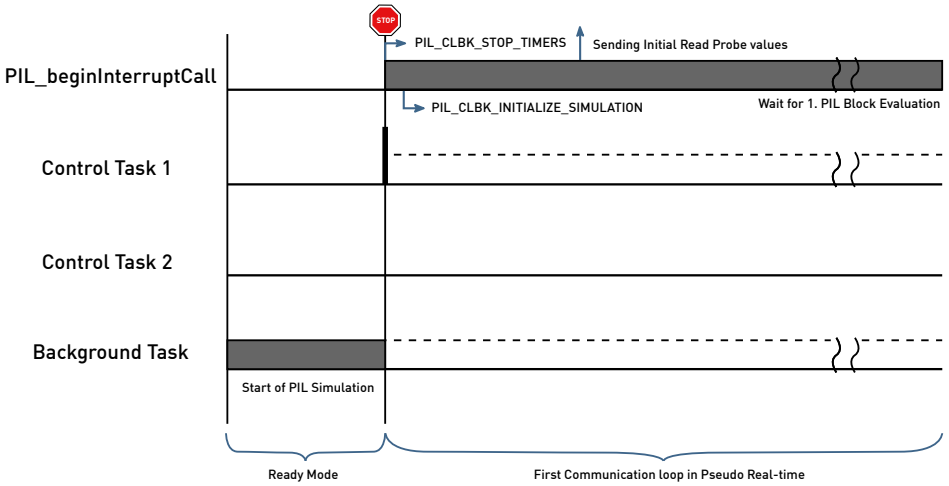
A target mode change can be requested either from the Target Manager or from the embedded application. Depending on the requested mode, the framework executes the appropriate callback. If the requested mode is equal to the current mode or while a PIL simulation is active, a mode request has no effect.

Target mode change requests are confirmed by the application code by calling the `PIL_allowPilSimulation()` and `PIL_inhibitPilSimualtion()` functions. Those functions also have no effect while a PIL simulation is active. Please refer to the example projects provided by Plexim for further details and implementation examples.

Simulation Start and Termination

When running multiple PIL simulations and comparing results it is important that all simulation-runs begin with identical initial conditions. This is

achieved by means of the `PIL_CLBK_INITIALIZE_SIMULATION` request, which is issued via the control callback at the beginning of a simulation.



Start of a PIL Simulation

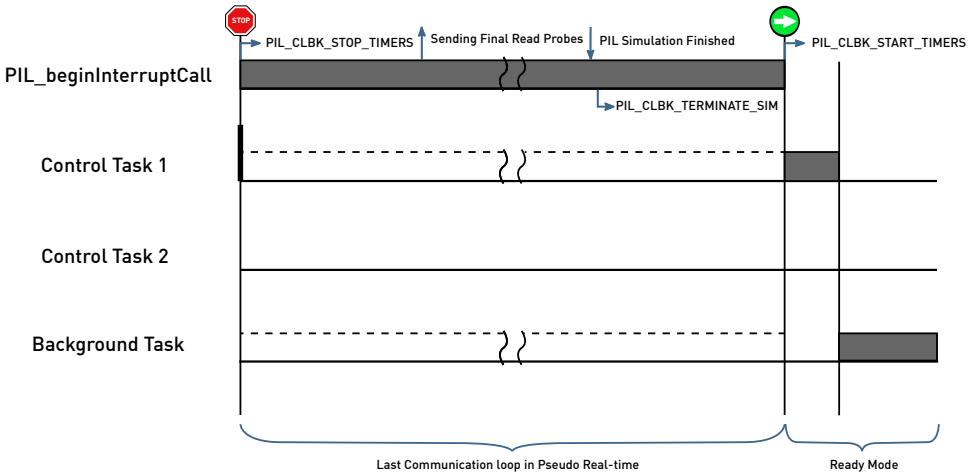
Note The initial conditions of Read Probes are fed into the PLECS model at simulation time t=0. However, these values will be immediately modified if the PIL block is also triggered at time t=0 and the output delay of the block is set to zero.

At the end of a PIL simulation, a `PIL_CLBK_TERMINATE_SIMULATION` request is issued prior to returning to real-time operation.

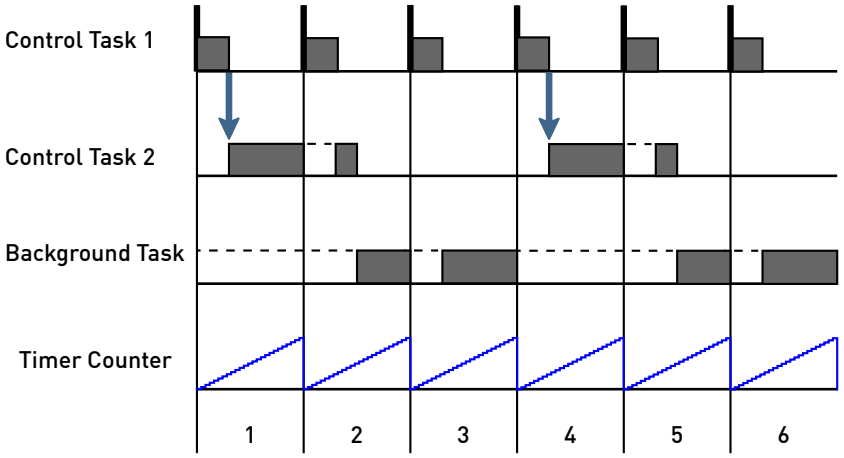
Control Dispatching

During a PIL simulation, the target operates in a pseudo real-time fashion with the execution of the control tasks being paced and synchronized with the simulation.

In the example shown in the next figure, the interrupt for Control Task 1 is based on the period of a hardware timer. Therefore, the timer period directly determines the amount of time available for the execution of the control tasks until the next interrupt occurs.



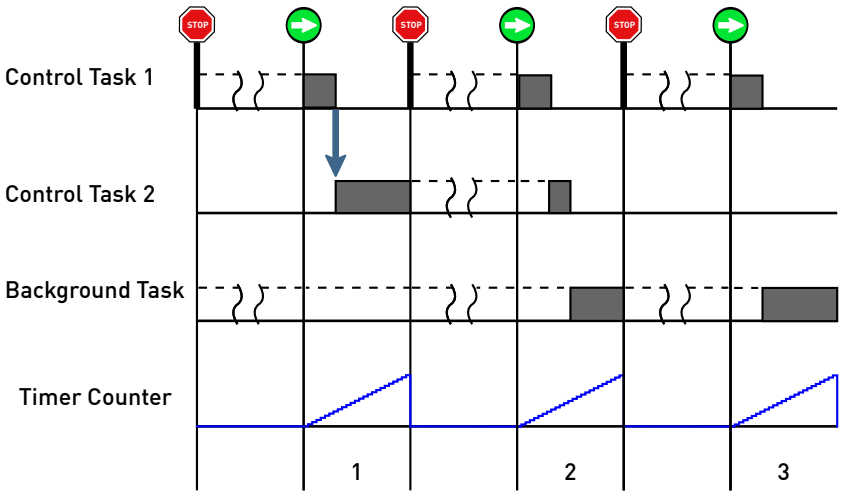
End of a PIL Simulation



Real-time operation with timer

To preserve the timing integrity in stepped mode, the hardware timer needs to be halted at the beginning of the communication loop and resumed when a step request is received, resulting in pseudo real-time operation.

By means of the CLBK_STOP_TIMERS and CLBK_START_TIMERS callback actions, the user is able to provide the necessary functionality specific to the actual application.



Pseudo real-time operation with periodically stopped timer

Task Synchronization at Start of Simulation

When control algorithms are distributed over multiple (nested) tasks, it is important to synchronize the start of a PIL simulation with the sequencing of the control tasks. In other words, after a PIL simulation has been started, a predictable and repeatable amount of time should elapse until the first execution of each nested task.

Such synchronization can be achieved by actively resetting the task dispatcher when the `PIL_CLBK_INITIALIZE_SIMULATION` request is received, as illustrated below.

Framework Configuration

The initialization and configuration of the PIL framework consists of three mandatory steps as well as a number of optional configurations.

- `PIL_init()` – Must be executed before any calls to the framework are made.
- `PIL_setLinkParams(...)` – Specifies the GUID to the framework and registers the interrupt callback for communication.
- `PIL_setCtrlCallback(...)` – Registers the control callback for PIL simulations.

Active task synchronization via simulation initialization callback

```
void PilCallback(PIL_CtrlCallbackReq_t aCallbackReq)
{
    switch(aCallbackReq)
    {
        case PIL_CLBK_INITIALIZE_SIMULATION:
            //application specific code
            ...
            //active synchronization of control task dispatching
            divider = TASK2PERIOD -1;
            break;
            .
            .
            .
        default:
            //catching an undefined callback
            break;
    }
}
```

```
    PIL_init();
    PIL_setLinkParams(\
        (unsigned char*)&PIL_D_Guid[0], \
        (PIL_CommCallbackPtr_t)SCIPoll
    );
    PIL_setCtrlCallback((PIL_CtrlCallbackPtr_t)PilCallback);
```

Optional configurations are as follows:

- `PIL_setNodeAddress(...)` – Configures node address for multi-drop serial communications.
- `PIL_setBackgroundCommCallback(...)` – Registers the background communication callback.

Configuration Constants

The `PIL_CONST_DEF` macro is used for making settings and diagnostics information available to PLECS. At a minimum, `Guid[]` must be defined. If a serial link is used for communication between PLECS and the target, then it is also necessary to specify to PLECS the communication rate by means of the

`BaudRate` definition. Optionally, further constants can be defined as shown below.

```
1      PIL_CONST_DEF(unsigned char, Guid[], CODE_GUID);
2      PIL_CONST_DEF(unsigned char, CompiledDate[], COMPILE_TIME_DATE_STR);
3      PIL_CONST_DEF(unsigned char, CompiledBy[], USER_NAME);
4
5      PIL_CONST_DEF(uint32_t, BaudRate, BAUD_RATE);
6      PIL_CONST_DEF(uint16_t, StationAddress, 0);
7      PIL_CONST_DEF(char, FirmwareDescription[], "Demo project");
```

Note Depending on the build settings it might be necessary to provide specific compiler/linker instructions (e.g. `#pragma RETAIN`) to prevent PIL definitions and constants that are not referenced by the code from being removed from the binary file.

Initialization Constants

The PIL framework also provides a mechanism to define "Initialization Constants" (or "Configurations") that can be read from the symbol file at the beginning of a simulation and used to configure the PLECS circuit.

`PIL_CONFIG_DEF` macro is used for defining such constants. They must be of integer or float type. Strings and arrays are not supported.

```
1      PIL_CONFIG_DEF(uint32_t, SysClk, SYSCLK_HZ);
2      PIL_CONFIG_DEF(uint32_t, PwmFrequency, PWM_HZ);
3      PIL_CONFIG_DEF(uint32_t, ControlFrequency, CONTROL_HZ);
4      PIL_CONFIG_DEF(uint16_t, ProcessorPartNumber, 28069);
```

To retrieve the values of the initialization constants in PLECS use the `plecs('get', '`*path to PIL block*`', 'InitConstants')` command either in a m-file or in the model initialization commands.

```
initConstants = plecs('get','./PIL','InitConstants');

Processor = initConstants.ProcessorPartNumber;
SysClk = initConstants.SysClk;
Fs = initConstants.ControlFrequency;
Fpwm = initConstants.PwmFrequency;
```

4

Microchip dsPIC33F Peripheral Models

Introduction

Microcontrollers (MCUs) for control applications typically contain peripheral modules such as Analog-to-Digital Converters (ADCs) and pulse width modulators (PWMs). These peripherals play an important role, since they act as the interface between the digital/analog signals of the control hardware and the control algorithms running on the processor. State-of-the-art MCUs often include peripherals with a multitude of advanced features and configurations to help implement complex sampling and modulation techniques.

When modeling power converters in a circuit simulator such as PLECS, it is desirable to represent the behavior of the MCU peripherals as accurately as possible. Basic Sample&Hold blocks and PWM modulators are useful for higher-level modeling. However, important details with regards to timing and quantization are lost when attempting to model an ADC with a basic zero-order hold (ZOH) block. For example, employing an idealized modulator to generate PWM signals can result in simulation results substantially different from the real hardware behavior.

Accurate peripheral models are even more important in the context of Processor-In-the-Loop (PIL) simulations. In this case, it is imperative to utilize peripheral models which are configurable exactly as the real implementations, i.e. by setting values in peripheral registers. By the same token, the inputs and outputs of the peripheral models must correspond precisely to the numerical representation in the embedded code.

The PLECS PIL library includes high-fidelity MCU peripheral models which work at the register level, and are therefore well-suited for PIL simulations. Furthermore, certain blocks have a second implementation with a graphical

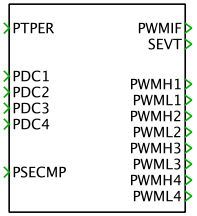
user interface (GUI) that automatically determines the register configurations based on text-based parameter selections.

Subsequent sections describe the PLECS peripheral components in detail and highlight modeling assumptions and limitations. When documenting peripheral register settings, the following color coding is used:

1. **Grey** (dark shading): No effect on the model behavior
2. **Green** (light shading): Register cell affects the behavior of the model

Microchip Motor Control PWM

The PLECS peripheral library provides two blocks for the Microchip Motor Control PWM (MCPWM) module, one with a register-based configuration mask and a second with a graphical user interface. The figure below shows the register-based version of the MCPWM module.



Register-based MCPWM module model

The register-based version allows the user to directly enter register values in decimal, binary, or hexadecimal notation. For convenience, the peripheral library also provides a component with a graphical user interface to simplify the configuration.

Both MCPWM blocks interface with other PLECS components over the following terminal groups:

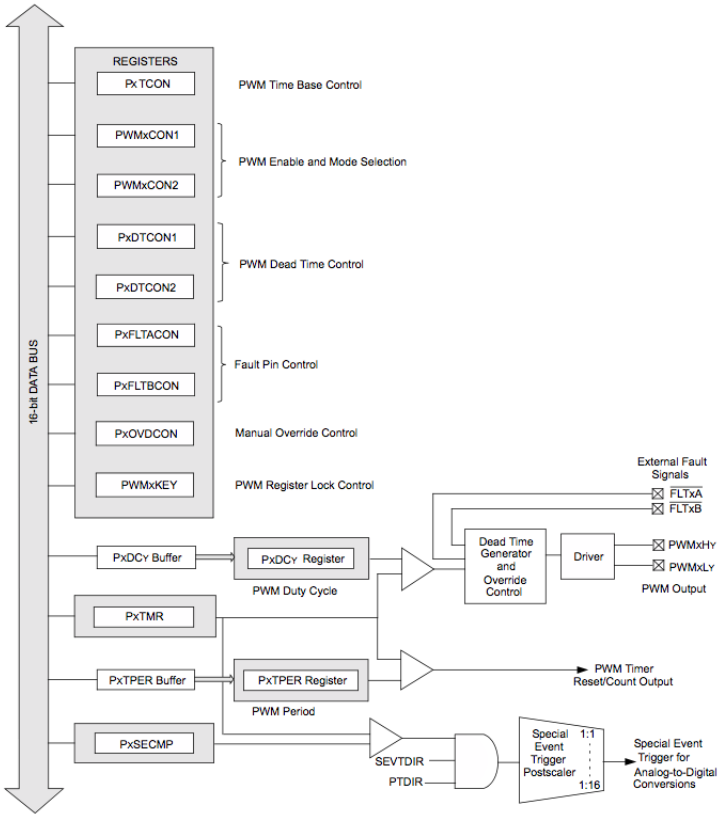
- *PDCx* - input ports for duty cycle register
- *PSECMP* - input port for special event trigger compare register
- *PWMIF* - output port for PWM interrupt flag
- *SEVT* - output port for special event trigger
- *PWMHx/Lx* - output ports for PWM signals

Note In the PLECS MCPWM module, PWM Faults and PWM Output Override have not been modeled

---

MCPWM Module Overview

The PLECS MCPWM model implements the most relevant features of the MCU peripheral.



Overview of the MCPWM module[1]

The MCPWM model implements the following features:

- PWM Clock Control
- PWM Output Control and Resolution
- Interrupt Control
- Special Event Trigger
- Dead Time Generator

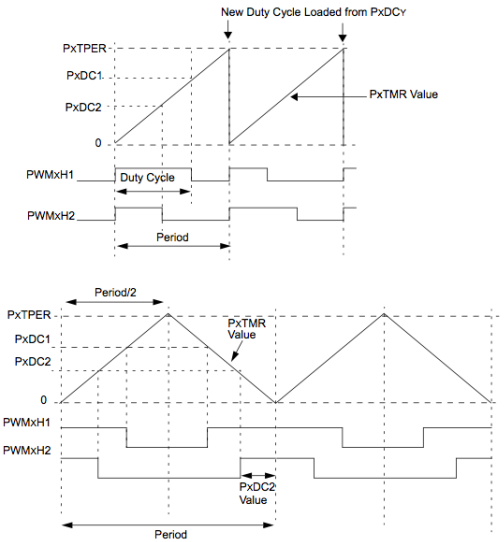
A section summarizing the differences of the PLECS MCPWM module as compared to the actual MCPWM module is provided in the “Summary” (on page 57) section.

PWM Clock Control

The modeled MCPWM realizes a counter that can operate in three different modes for the generation of asymmetrical and symmetrical PWM signals. The three supported modes are:

- *Free Running mode*
- *Continuous Up/Down mode*
- *Continuous Up/Down mode with interrupts for double PWM updates*

The counter for these modes is visualized below.



Counter modes [1]

In *Free Running mode*, the counter is incremented from 0 to a counter period *PTPER* using a counter clock operated at a clock frequency of F_{CY}. The *PTPER* value corresponding to a desired PWM frequency (F_{PWM}) can be calculated as:

$$PTPER = \frac{F_{CY}}{F_{PWM} \cdot PTMR\,Prescaler} - 1$$

When the counter reaches the period (*PTPER*), the subsequent count value is reset to zero, duty cycle (*PDCx*) and special event (*PSECMP*) registers are updated, and the sequence is repeated.

In the *Continuous Up/Down mode*, and *Continuous Up/Down mode with interrupts for double PWM updates*, the counter is incremented from 0 to a counter period *PTPER* and then decremented back to 0 using a counter clock operated at a clock frequency of F_{CY}. The *PTPER* value corresponding to a desired PWM frequency (F_{PWM}) can be calculated as:

$$PTPER = \frac{F_{CY}}{2 \cdot F_{PWM} \cdot PTMR\ Prescaler} - 1$$

In the *Continuous Up/Down mode*, when the counter reaches 0, the duty cycle (*PDCx*) and special event (*PSECMP*) registers are updated.

In the *Continuous Up/Down mode with interrupts for double PWM updates*, the duty cycle (*PDCx*) and special event (*PSECMP*) registers are updated when the counter reaches 0 and *PTPER*.

Note In the PLECS MCPWM module, *Single Event Mode* is not allowed.

While the system clock and the period counter value are separately defined in the mask parameters, the counter mode and the clock divider are jointly configured in the *PTCON* register.

PTCON Register Configuration [1]

The input clock (T_{CY}) derived from the oscillator source can be prescaled using the *PTCKPS* bits in the *PTCON* register. Additionally, the counter mode selected using the *PTMOD* bits and the time-based output post scalar (*PTOPS*) bits determine the generation of the PWM interrupt flag.

Example Configuration – Step 1

This example shows the configuration of the PWM module operating in *Free Running mode* with a 50 μs period. The *PTCON* register is configured to:

$$PTCON = 4 \;\hat{=}\; 0\,0\,0\,0\,0\,0\,0\,0\,\underbrace{0\,0\,0\,0}_{PTOPS}\;\underbrace{0\,1}_{PTCKPS}\;\underbrace{0\,0}_{PTMOD}$$

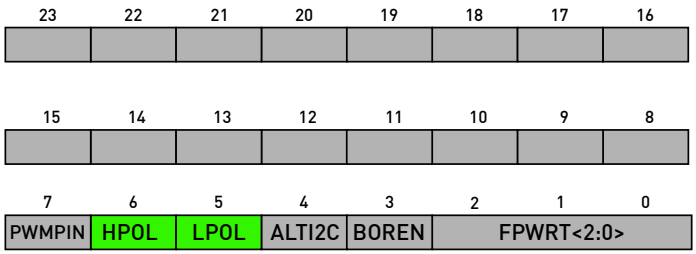
According to this configuration, the time-based submodule is operating in the *Free Running mode* with a timer clock period four times the system clock period. For a *PTPER* value of 999 and an 80 MHz system clock, the resulting PWM signal has the following period:

$$T_{PWM} = (PTPER + 1) \cdot \frac{PTCKPS}{F_{CY}} = 50\ \mu s.$$

PWM Output Control and Resolution

The MCPWM model for a non-zero duty cycle results in outputs of the PWM generators to be driven active at the beginning of the PWM period. Each PWM output will be driven inactive when the value of the counter matches the duty cycle value of the PWM generator. If the value of the duty cycle register is zero, the output on the corresponding PWM pin is inactive for the entire PWM period. The PWM output is active for the entire period if the value of *PDC* is greater than *PTPER*.

Note In the implemented model, immediate update of the *PDC* and *PSECMP* registers is not modeled.



FPOR:POR Register Configuration [1]

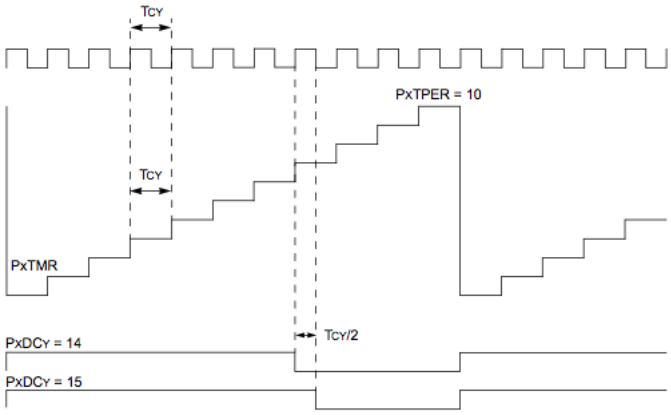
The *HPOL* and *LPOL* bits in the *FPOR:POR* register determine the output polarity of the high-side and low-side output pins of the PWM generators. For example, if the *LPOL* bit is set, then the low-side output is high when the PWM is active and low when the PWM is inactive. If the bit is cleared, then the low-side output is low when the PWM is active and high when the PWM is inactive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				PMOD4	PMOD3	PMOD2	PMOD1	PEN4H	PEN4L	PEN3H	PEN3L	PEN2H	PEN2L	PEN1H	PEN1L

PWMCON1 Register Configuration [1]

In the MCPWM, each PWM generator can be operated in either complementary or independent mode. In complementary mode both output pins cannot be active simultaneously. Additionally, a dead time is inserted during device switching making both outputs inactive for a short period. In independent mode there are no restrictions on the state of the pins for a given output pin pair. Additionally, the dead time module is disabled when the PWM module is operated in independent mode. The mode for each of the PWM generators is selected by configuring the bits *PMOD4:PMOD1* in the *PWMCON1* register.

The first bit of the register *PDC* determines whether the PWM signal edge occurs at the T_{CY} or $\frac{T_{CY}}{2}$ boundary. The figure below illustrates the effect of this bit on the PWM output.

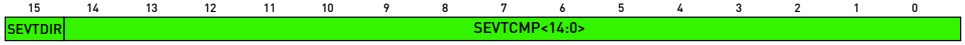


Duty cycle resolution timing diagram, Free Running mode, and 1:1 prescaler selection [1]

Special Event Trigger

The MCPWM can be configured to trigger the Analog-to-Digital (ADC) converter using the special event compare register (*PSECMP*). This allows ADC sampling and conversion timing to be synchronized to the PWM time base and

provides the flexibility of programming the start of conversion at any point within the PWM period.



PSECMP Register Configuration [1]

The PWM counter register is compared to the *SEVTCMP* bits in the *PSECMP* register and generates a trigger signal when the counter value is equal to the *SEVTCMP* bits. In *Up/Down Count mode*, the *SEVTDIR* bit provides added flexibility on the generation of the trigger signal. When this bit is set, the trigger is generated on a match event when the counter is counting down. When the bit is set to zero, the trigger is generated on a match event when the counter is counting up.

Additionally, the Special Event Trigger Postscaler (*SEVOPS*) bits in the *PWMCON2* register allows a 1:1 to 1:16 post scale ratio. These bits can be configured if the ADC conversions are not required every PWM cycle.

PWMCON2 Register Configuration [1]

Interrupt Control

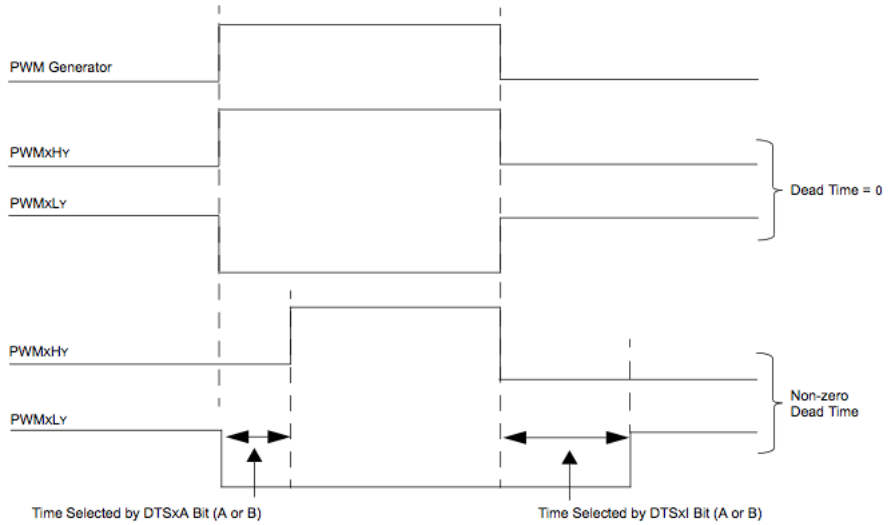
The MCPWM module can be configured to generate an interrupt flag depending on the mode of operation and the time base postscaler (*PTOPS*) bits in the *PTCON* register. In the model the interrupt flag (*PWMIF*) is internally reset automatically after one simulation step.

In the *Continuous Up/Down mode with interrupts for double PWM updates*, an interrupt event is generated each time the counter equals 0 and *PTPER*. The postscaler selection bits are ignored in this mode.

In the *Free Running mode* the interrupt flag is generated when the counter is reset to 0. In the *Continuous Up/Down mode*, the interrupt flag is generated when the counter is equal to 0 and the counter is counting up. In both of these modes, the postscaler bits can be used to reduce the frequency of interrupt events.

Dead Time Generator

In independent mode, the dead-time module is inactive and no dead-time is inserted between the high-side and low-side PWM signals of a PWM output generator. When operated in complementary mode, each PWM output generator can be configured to have some dead time between the turn on and turn off of the high-side and low-side PWM signals.



Dead time insertion [1]

The Dead Time Control Register 1 (*PDCTON1*) is used to configure two different dead-time units (Unit A and Unit B). The *DTA* bits are used to assign a 6-bit dead-time value for Unit A. The *DTAPS* bit is used to configure the dead-time clock as a multiple of the system clock (T_{CY}). The corresponding bits *DTB* and *DTBPS* are used to configure Unit B.

PDTCON1 Register Configuration [1]

The dead-time for Unit A and Unit B, are calculated as follows:

$$Dead\ Time = (DTx + 1) \cdot T_{CY} \cdot DTxPS,$$

where x refers to Unit A or B.

The Dead Time Control Register 2 (*PDCTON2*) contains configuration bits that are used to control the insertion of dead time when the high-side or low-side PWM signals become active. The *DTS1I* - *DTS4I* bits select the dead time inserted before *PWML1* - *PWML4*, respectively, are driven active. The *DTS1A* - *DTS4A* bits select the dead time inserted before *PWMH1* - *PWMH4*, respectively, are driven active.

PDTCON2 Register Configuration [1]

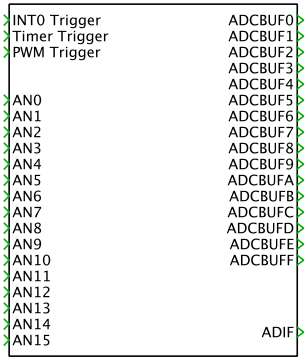
Summary of PLECS Implementation

The PLECS MCPWM module models the major functionality of the actual MCPWM module. Below is a summary of differences of the PLECS MCPWM module compared to the actual MCPWM module:

- PWM Faults and PWM Output Override are not supported.
- *Single Event Mode* is not supported.
- Immediate update of the *PDC* and *PSECMP* registers is not supported.
- PWM update lockout is not supported.
- The interrupt flag (*PWMIF*) is internally reset automatically after one simulation step.

Microchip Motor Control ADC

The PLECS peripheral library provides two blocks for the Microchip Motor Control ADC (MCADC) module, one with a register-based configuration mask and a second with a graphical user interface. The figure below shows the appearance of the register-based version.



Register-based MCADC module model

The register-based version allows the user to directly enter register values in decimal, binary, or hexadecimal notation. For convenience, the peripheral library also provides a component with a graphical user interface to simplify the configuration.

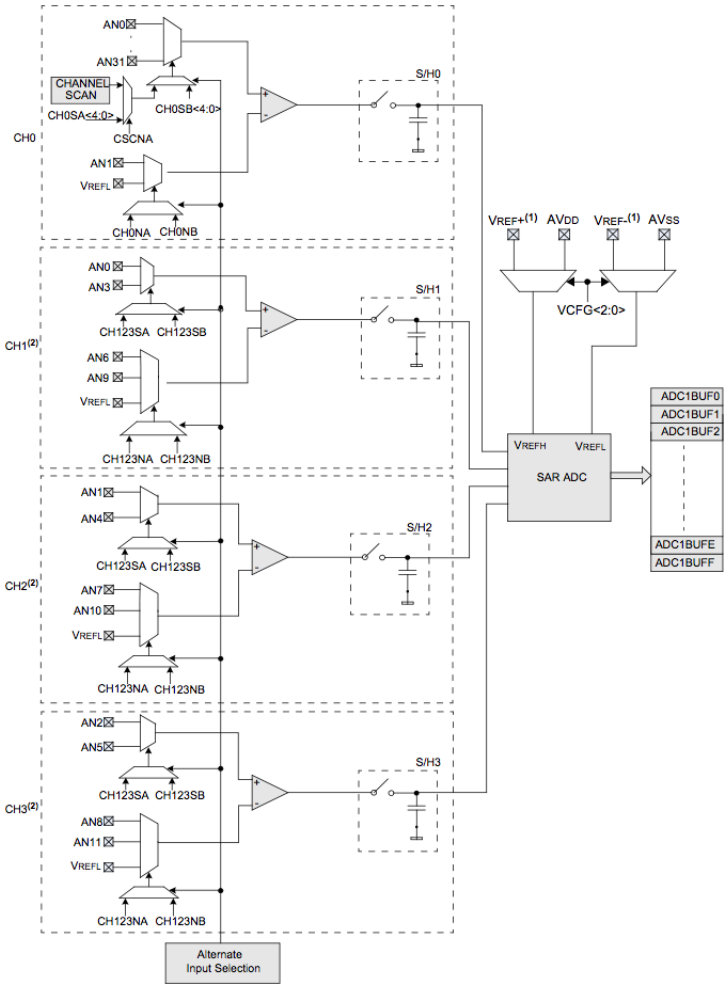
Both MCADC blocks interface with other PLECS components over the following terminal groups:

- *ANx* - input ports for duty cycle register
- *Triggers* - input port for INT0, Timer, and PWM triggers
- *ADCBUFx* - output port for ADC buffer register
- *ADIF* - output port for ADC interrupt flag

Note In the PLECS MCADC module, the GP timer triggers (Timer 3 and Timer 5) and Motor Control PWM 1 and 2 triggers have been lumped into a single Timer and PWM trigger, respectively.

MCADC Module Overview

The PLECS MCADC model implements the most relevant features of the MCU peripheral.



Overview of the MCADC module without DMA [2]

The MCADC model implements the following features:

- ADC Configuration

- ADC Sampling and Conversion
- Multi-channel ADC Sampling Mode
- ADC Input Selection Mode
- ADC Interrupt Logic
- ADC Buffer Fill Mode

A section summarizing the limitation of the PLECS MCADC module as compared to the actual MCADC module is provided in the “Summary” (on page 68) section.

ADC Configuration

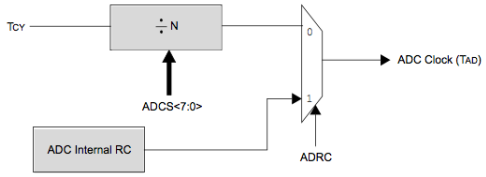
The MCADC module can be operated either in 10-bit or 12-bit operation mode. The 12-bit Operation Mode bit (*AD12B*) in the *ADCON1* register allows the ADC module to function as either a 10-bit, 4-channel ADC (when the *AD12B* bit is cleared) or a 12-bit, single-channel ADC (when the *AD12B* bit is set). In 10-bit mode, the *CHPS* bits in the *ADCON2* register can be configured to operate the MCADC module to convert:

- only $CH0$
- $CH0$ and $CH1$
- $CH0$, $CH1$, $CH2$, and $CH3$

The *VCFG* bits in the *ADCON2* register allow the selection of the voltage references for the MCADC module. The voltage reference high (V_{REFH}) and the voltage reference low (V_{REFL}) for the ADC module can be supplied from the internal AV_{DD} and AV_{SS} voltage rails or the external V_{REF+} and V_{REF-} input pins. The table below summarizes the different configurations that are possible by setting the *VCFG* bits.

VCFG	V_{REFH}	V_{REFL}
000	AV_{DD}	AV_{SS}
001	AV_{DD}	V_{REF-}
010	V_{REF+}	AV_{SS}
011	V_{REF+}	V_{REF-}
1xx	AV_{DD}	AV_{SS}

The MCADC module clock (T_{AD}) can be configured to use the system clock (T_{CY}) or a dedicated internal RC clock (T_{ADRC}). The figure below summarizes the generation of the ADC clock.



ADC Clock Generation [2]

While the system clock and the period counter value are separately defined in the mask parameters, the ADC clock source selection (*ADRC*) and the clock divider (*ADCS*) are jointly configured in the *ADCON3* register.

ADCON3 Register Configuration [2]

The clock divider is used to lower the frequency when the ADC clock is derived from the system clock. The *ADCS* bits allow the clock to be scaled to one of 64 settings, from 1:1 to 1:64. The table below summarizes the effect the *ADCS* and *ADRC* bits have on the ADC clock period.

ADRC	ADC Clock Period (T_{AD})
0	$T_{CY} \cdot (\text{ADCS} + 1)$
1	T_{ADRC}

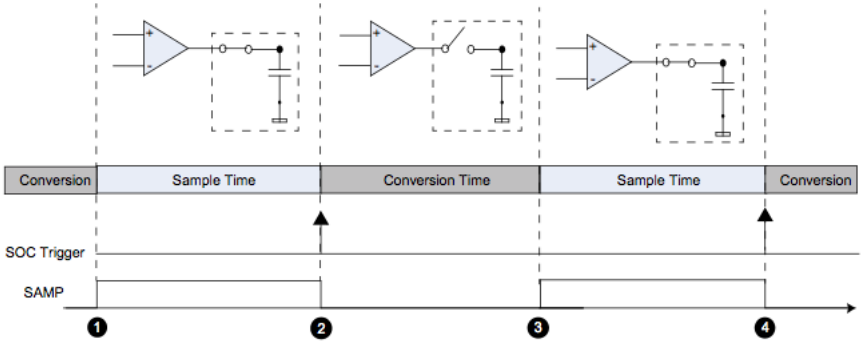
Note *ADCS* values over 63 are reserved in the actual hardware and will be flagged as an error in the PLECS MCADC module.

The MCADC module can be configured to output the ADC results in four different numerical formats. The *FORM* bits in the *ADCON1* register select the data format. Further, in the PLECS MCADC module the output format can be configured as quantized double format for convenience. The Output mode

block parameter selects if the *FORM* bits are used or if the output is presented as a quantized double format. The table below summarizes the different available formats.

FORM	Output Mode	Data Format
00	Use FORM bits	Unsigned Integer
01	Use FORM bits	Signed Integer
10	Use FORM bits	Unsigned Fractional
11	Use FORM bits	Signed Fractional
xx	Quantized Double	Quantized Double

ADC Sampling and Conversion



Note 1: Sampling starts automatically after conversion.
2: Conversion starts upon trigger event.
3: Sampling starts automatically after conversion.
4: Conversion starts upon trigger event.

Automatic Sample and Triggered Conversion Sequence [2]

The actual MCADC module can be configured to operate in different modes. Below is a list of the possible configurations for the actual MCADC:

- Manual Sample and Manual Conversion Sequence
- Manual Sample and Automatic Conversion Sequence
- Manual Sample and Triggered Conversion Sequence

- Automatic Sample and Manual Conversion Sequence
- Automatic Sample and Automatic Conversion Sequence
- Automatic Sample and Triggered Conversion Sequence

In the PLECS MCADC module only the Automatic Sample and Triggered Conversion Sequence mode has been modeled. The figure above summarizes the operation of this mode.

In this mode, the sampling of the channels starts automatically after a conversion is completed. Automatic sampling is enabled by setting the *ASAM* bit in the *ADCON1* register. The conversion is started upon trigger event from one of the external SOC trigger sources. This allows ADC conversion to be synchronized with the internal or external events. The external trigger source is selected by configuring the *SSRC* bits to

- 001 when using External Interrupt Trigger
- 010 or 100 when using Timer Interrupt Trigger
- 011 or 101 when using Motor Control PWM Special Event Trigger

---

Note In the PLECS MCADC module, clearing the *ASAM* bit is not allowed. This bit must always be set. Additionally, in the actual hardware the ADC module takes some time to stabilize. There is no such requirement in the implemented MCADC module.

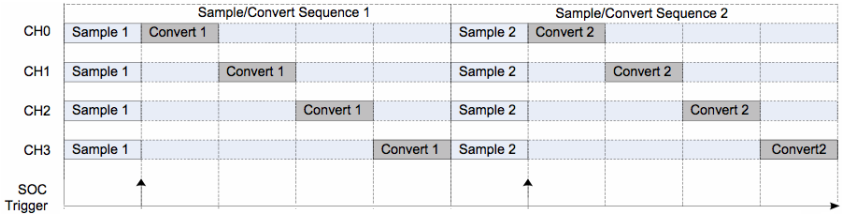
The MCADC can be operated either as a single-channel 12-bit or multi-channel 10-bit module. The time required to complete a conversion (T_{CONV}) is dependent on whether the ADC is operated in 12-bit or 10-bit mode. The table below summarizes the amount of time required to completed one conversion in the two modes:

Mode	T_{CONV}
10-bit	$12 \cdot T_{AD}$
12-bit	$14 \cdot T_{AD}$

Multi-channel ADC Sampling Mode

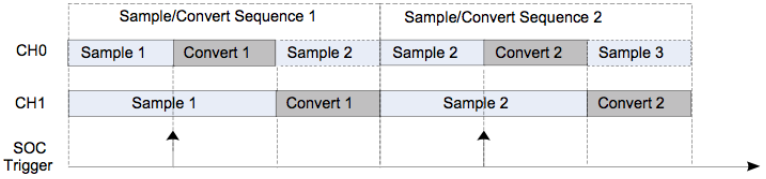
The MCADC works as single channel converter when operated in as a 12-bit ADC module. In this mode the inputs to $CH1$, $CH2$, and $CH3$ are ignored and

only $CH0$ is converted. When operated as a 10-bit ADC module, the MCADC can be configured to operate as a multi-channel ADC module. In the multi-channel operation, the MCADC module can be configured to operate in simultaneous or sequential sampling modes. In simultaneous sampling mode, the sampling of all channels is stopped when an SOC trigger is received. The figure below shows the timing diagram of a 4-channel module operated with simultaneous sampling in the Automatic Sample and Triggered Conversion Sequence mode.



4-Channel Simultaneous Sampling [2]

When the multi-channel ADC module is operated in sequential mode, the sampling for $CH0$ ends when an SOC trigger is received. The sampling of $CH1$ ends once the conversion of $CH0$ is completed. The same logic applies to the end of sampling for $CH2$ and $CH3$. The figure below shows the timing diagram of a 2-channel module operated with sequential sampling in the Automatic Sample and Triggered Conversion Sequence mode.

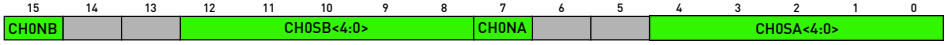


2-Channel Sequential Sampling [2]

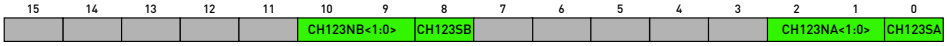
Note Any SOC trigger received while the MCADC module is converting will be lost. Conversions are started when an SOC trigger is received while the module is sampling all active channels.

ADC Input Selection Mode

The *ADCHS0* and *ADCHS123* registers are used to configure which analog input channels are selected as the positive and negative input selections for $CH0$, and $CH1$, $CH2$, and $CH3$, respectively. The figures below show the two registers:



ADCHS0 Register Configuration [2]



ADCHS123 Register Configuration [2]

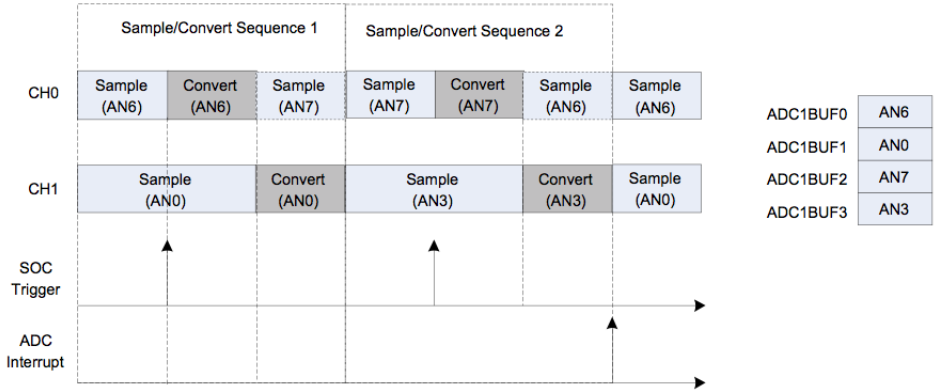
In the MCADC module, each channel can be configured to operate in fixed input selection mode which uses only MUXA, or in alternate input selection mode where both MUXA and MUXB are used. The table below summarizes the effect of the control bits on the analog input selection for each channel.

		MUXA		MUXB	
		Control bits	Analog Inputs	Control bits	Analog Inputs
CH0	+ve	CH0SA<4:0>	AN0 to AN31	CH0SB<4:0>	AN0 to AN31
	-ve	CH0NA	VREF-, AN1	CH0NB	VREF-, AN1
CH1	+ve	CH123SA	AN0, AN3	CH123SB	AN0, AN3
	-ve	CH123NA<1:0>	AN6, AN9, VREF-	CH123NB<1:0>	AN6, AN9, VREF-
CH2	+ve	CH123SA	AN1, AN4	CH123SB	AN1, AN4
	-ve	CH123NA<1:0>	AN7, AN10, VREF-	CH123NB<1:0>	AN7, AN10, VREF-
CH3	+ve	CH123SA	AN2, AN5	CH123SB	AN2, AN5
	-ve	CH123NA<1:0>	AN8, AN11, VREF-	CH123NB<1:0>	AN8, AN11, VREF-

When operated in fixed input selection mode, chosen by setting the *ALTS* bit in the *ADCON2* register to zero, only MUXA and the associated control bits are used to select the positive and negative analog inputs for each channel. When operated as a 12-bit module, only $CH0$ is sampled.

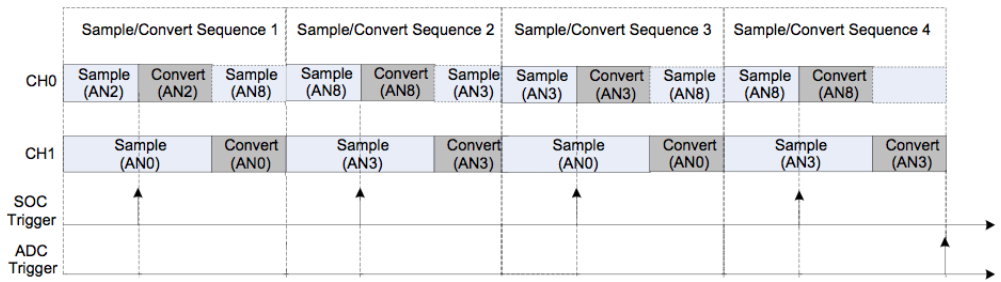
When operated in alternate input selection mode, chosen by setting the *ALTS* bit in the *ADCON2* register to 1, both MUXA and MUXB are used to select the positive and negative analog inputs for each channel. Again, when operated as a 12-bit module, only $CH0$ is sampled. In this mode the ADC completes one sweep using the MUXA selection and uses the MUXB selection in

the next sweep. In the next sweep MUXA is used again. This switch between MUXA and MUXB continues while the ADC is operated in this mode. The figure below shows the operation of a 2-channel module with alternate input selection in sequential sampling mode. The interrupt has been configured to occur after 4 conversions.



2-Channel Sequential Sampling in Alternate Input Selection mode [2]

The MCADC module provides further flexibility by allowing $CH0$ to be operated in scan mode. The Channel Scanning mode is enabled by setting the Channel Scan bit (*CSCNA*) in the *ADCON2* register.



2-Channel Sequential Sampling in Alternate Input Selection mode with Channel Scan enabled [2]

The desired conversion sequence is selected by configuring the appropriate bits in the channel selection register (*AD1CSSL*). The conversions are carried out in ascending order. If operated in alternate input selection mode with channel scan enabled, MUXA software control is ignored for $CH0$ and the

ADC module converts the first selected analog input. In the next sweep, the inputs selected by MUXB are measured. In the following sweep the next selected analog input is sampled for $CH0$. Input selections for $CH1$, $CH2$, and $CH3$ are unaffected. The figure above shows an example of a 2-channel sequential sampling module operated in alternate input selection mode with channel scanning enabled. $AN2$ and $AN3$ have been selected for channel scanning and $AN8$ has been selected by the MUXB input selector for $CH0$. An interrupt is generated after 8 conversions.

ADC Interrupt Logic

CHPS	SIMSAM	SMPI	Conversions per Interrupt	Description
00	x	N-1	N	1-Channel mode
01	0	N-1	N	2-Channel, Sequential Sampling mode
1x	0	N-1	N	4-Channel, Sequential Sampling mode
01	1	N-1	$2 \cdot N$	2-Channel, Simultaneous Sampling mode
1x	1	N-1	$4 \cdot N$	4-Channel, Simultaneous Sampling mode

The PLECS MCADC module reflects the properties of an actual MCADC module without DMA. The ADC module writes the results of the conversions into the analog-to-digital result buffer as conversions are completed. The *SMPI* bits in the *ADCON2* register determine the number of conversions for the MCADC module before an interrupt is generated. The results are written into the ADC buffer after each conversion is completed. The MCADC module supports 16 result buffers. Therefore, the maximum number of conversions per interrupt must not exceed 16.

The number of conversions per ADC interrupt depends on the following parameters, which can vary from one to 16 conversions per interrupt:

- Number channels selected
- Sequential or Simultaneous Sampling
- Samples Convert Sequences Per Interrupt bits (*SMPI*) settings

The table above summarizes the effect each of these factors has on the number of conversions per interrupt.

ADC Buffer Fill Mode

When the Buffer Fill Mode bit (*BUFM*) in the *ADCON2* register is set, the 16-word results buffer is split into two 8-word groups: a lower group (ADC1BUF0 through ADC1BUF7) and an upper group (ADC1BUF8 through ADC1BUFF). The 8-word buffers alternately receive the conversion results after each ADC interrupt event. When the *BUFM* bit is set, each buffer size equals eight. Therefore, the maximum number of conversions per interrupt must not exceed 8. When the *BUFM* bit is cleared, the complete 16-word buffer is used for all conversion sequences.

Summary of PLECS Implementation

The PLECS MCADC module models the major functionality of the actual MCADC module. Below is a summary of differences of the PLECS MCADC module compared to the actual MCADC module:

- The PLECS MCADC module models the Microchip MCADC module without DMA.
- The GP timer triggers (Timer 3 and Timer 5) and the Motor Control PWM 1 and 2 triggers have been lumped together into single Timer and PWM trigger, respectively.
- *ADCS* values over 63 in the *ADCON3* register will be flagged as an error in the PLECS MCADC module.
- Only Automatic Sample and Triggered Conversion Sequence mode is supported by the PLECS MCADC module. Clearing the *ASAM* bit in the *ADCON1* register will be flagged as an error.
- The PLECS MCADC module does not require any time for stabilization during startup.
- Any SOC trigger received while the MCADC module is converting will be lost. Conversions are started when an SOC trigger is received while the module is sampling all active channels.
- The output results are provided according to the numerical format specified by the *FORM* bits in the *ADCON1* register or as quantized double values.

Reference

1 - Pictures provided with Courtesy of Microchip, Literature Source: *Motor Control PWM Reference Guide*, Literature Number DS70187E, February 2007 - Revised September 2012

2 - Pictures provided with Courtesy of Microchip, Literature Source: *Motor Control ADC Reference Guide*, Literature Number DS70183D, December 2006 - Revised April 2012

5

Embedded Application

This chapter provides additional information about the dsPIC “FOC” demo application.

Importing the MPLAB X Demo Project

The source code of the embedded demonstration project is provided as part of the PIL Framework installation and can be directly opened as a project in MPLAB X.

Configuring the Project

The building of the demo project is configured to include a custom *pre-build* action.

At the start of a build, the PIL Prep Tool is called to generate the auxiliary symbols used by PLECS, as explained in “PIL Prep Tool” (on page 26).

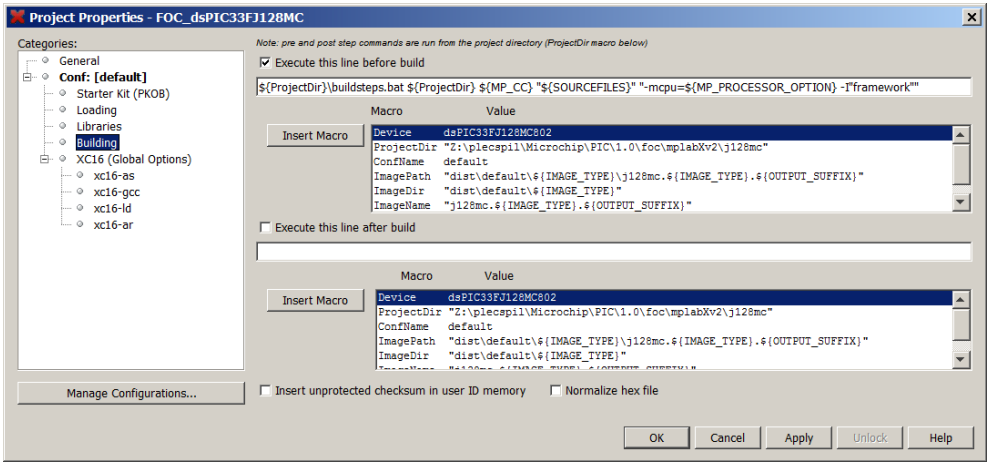


Figure 5.1: Pre-build step

This addtional build step is configured under **Project Properties** in the **Building** section.

Rebuilding the Project

The project can be compiled and flashed by clicking the **Make and Program Device** button on the toolbar or selecting **Run Project** from the **Run** menu.

After reflashing the dsPIC with your own project, make sure that the PLECS target manager is pointing to the correct symbol file (located in the `dist/default/production` folder).

Project Structure

The following is a brief description of the files which make up the embedded demo application.

Initialization and Task Dispatching

The following files contain the routines for the initialization of the core, setup of timers, hardware interrupts, software interrupts and tasks.

- `main.c/h` – `main()`-routine, hardware interrupt routine.
- `init.c/h` – Initialization of system clock, I/O and peripherals.

Control Law

The FOC control algorithms include the following functionality:

1. Measurement of phase currents and transformation into dq-frame.
2. Synchronous frame current control with decoupling, output saturation and anti-windup.
3. Space-vector modulation with voltage compensation.

The files related to the control algorithms are the following:

- `calib.c/h` – Control calibrations (settings).
- `pu.c/h` – Fixed-point reference values.
- `control.c/h` – Control tasks.
- `plx_control.lib` – Fixed-point control library:
  - `plx_types.h` – Type definitions
  - `fp_math.h` – Fixed-point math header file
  - `pidq.h` – Synchronous frame PI controller
  - `vector.h` – Park transformations
  - `svpwm.h` – Space vector modulation

In addition, the `modules` folder contains the header files for the fixed-point control library.

Communication Interface

The demo project utilizes the universal asynchronous receiver/transmitter (UART) interface for exchanging information with PLECS. The following files contain the code related to the UART interface:

- `init.c` – Initialization of peripheral and io.
- `main.c` – Communication callback function `PollUart()`

PIL Functionality

These are the files that are enabling the demo application for PIL simulation with PLECS.

- `pil.h` – PIL framework API.
- `pil_ctrl.c/h` – Control callback for stepping the control tasks during a PIL simulation. See “Control Callback” (on page 38).
- `pil_symbols_c.inc/c` – Defines PIL constants according to “Configuration Constants” (on page 44).
- `pil_symbols_p.inc/c` – Definitions of override and read probe attributes. See “Probes” (on page 26).
- `pil_framework_lib.lib` – PIL framework library.

Note The files `pil_symbols_p.inc` and `pil_symbols_c.inc` are generated by the PIL Prep Tool and should therefore not be edited or revision controlled.

---

plexim

electrical engineering software

Plexim GmbH info@plexim.com www.plexim.com