THE SIMULATION PLATFORM FOR POWER ELECTRONIC SYSTEMS

PIL-FOC Demo for C2000 MCUs Version 3.6

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 C2000 MCUs

Contents

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 3.6 or higher)
- Code Composer Studio (CCS) v5.5 – Download from ti.com.
- C2Prog – Download from codeskin.com.

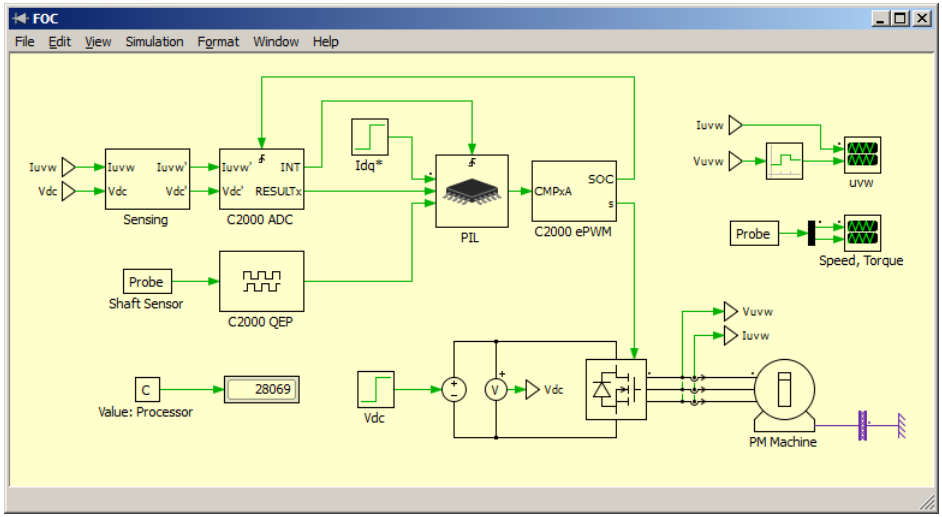
A license is required to run PLECS and activate the PIL package. You can request such a license from Plexim at plexim.com. Copy the license file `license.dat` that will be supplied to you into the directory in which you have installed PLECS.

1

Getting Started

This chapter provides a hands-on demonstration of how control-code executing on a TI C2000 Piccolo or Delfino 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 are provided in 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 28069 controlSTICK or 28335 controlCARD with serial communication over USB.

Configuring the Hardware

28069 controlSTICK

No special configurations are required.

28335 controlCARD

SW1 must be in the “OFF” position, pointing to the connector side of the controlCARD to allow serial communicaton over the FTDI chip.

All SW2 switches should be “ON”, pointing towards the upper edge of the card.

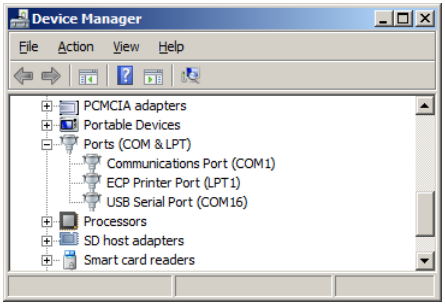
Docking Station

J9 must be closed.

Loading the Firmware

Connect the JTAG/SCI USB port to your PC. Open the Windows Device Manager and confirm the enumeration of a COM port.

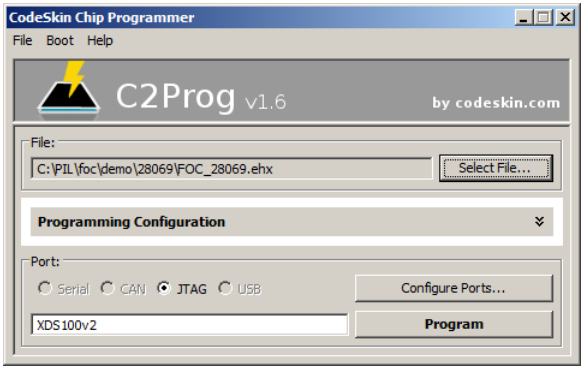
You may have to install the FTDI drivers if the port is not enumerated.



COM port listed in device manager

Depending on your hardware, either of the pre-compiled executables `F0C_28069.ehx` or `F0C_28335.ehx` must be loaded.

In C2Prog, select the ehx-file and configure the port to `XDS100v2`.



Flashing the controlSTICK

Click the **Program** button.

Once the reflashing completes, power-cycle the processor. Confirm that the following LED is blinking:

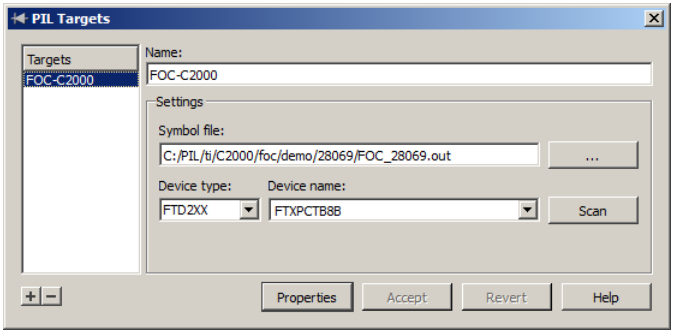
- On 28069 controlSTICK: LD2
- On 28335 controlCARD: Red LED in the corner of the board

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**.



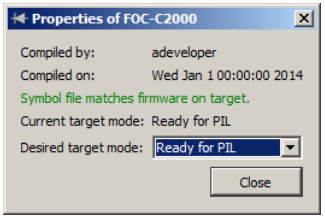
Target configuration

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 TI codegen tools. Select `FOC_28069.out` or `FOC_28335.out`, depending on your hardware.

The remaining target configuration is the communication link. Select **FTD2XX** from the **Device type** combo box. Then click on **Scan** and select the second port (typically ending with B) 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.



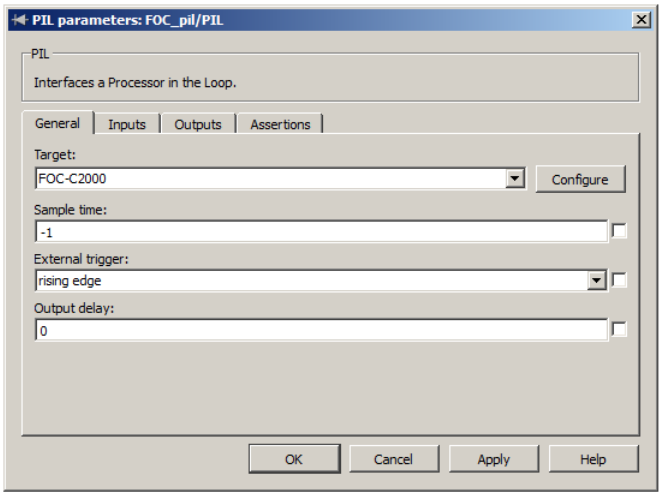
Target properties

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.

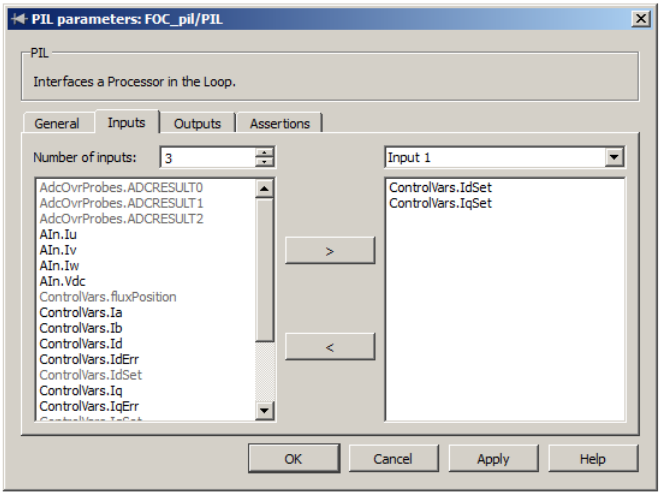
Notice how the PIL block has been configured for an external trigger input. This allows the execution of the PIL block and associated embedded control code to be triggered by the ADC end-of-conversion (EOC) event. The ADC, in turn, is triggered by an ePWM start-of-conversion (SOC) event in this example.



PIL block general configuration

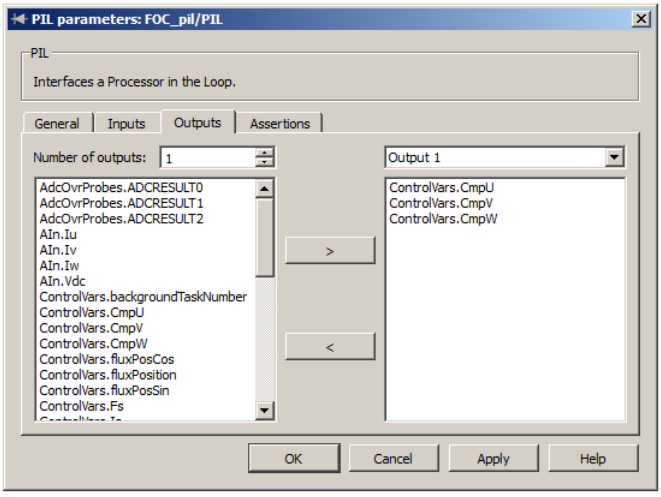
Activate the **Inputs** tab and see how the PIL block has been configured for the following three (3) inputs.

- `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.



PIL block inputs

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 outputs

The PIL block has been further configured for one (1) output (**Outputs** tab):

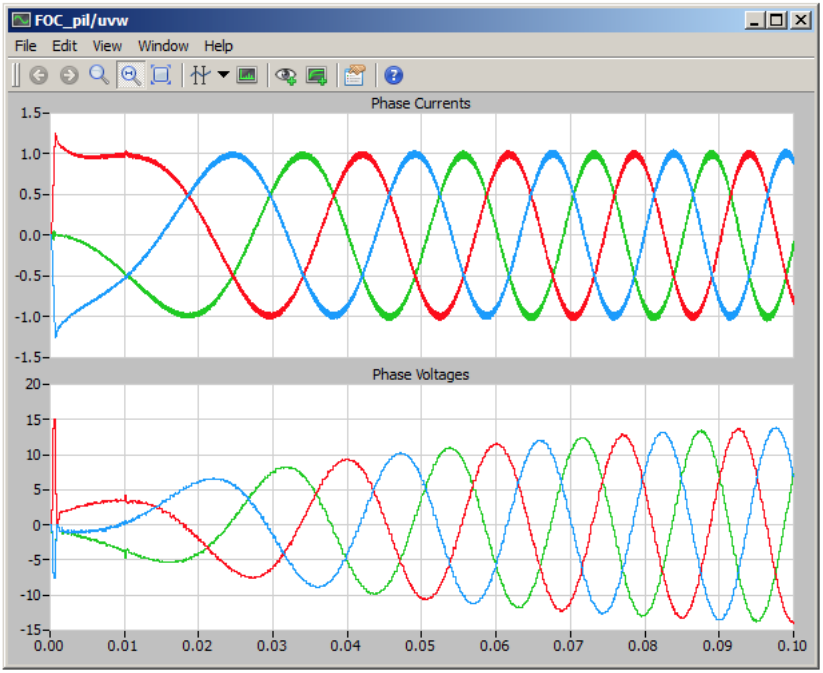
- `ControlVars.ePWM_CmpU,V,W` – ePWM peripheral compare register values.

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.

Running the PLECS Model

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

Observe how the embedded control algorithm is maintaining tight current regulation as the motor accelerates.



PIL simulation result

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

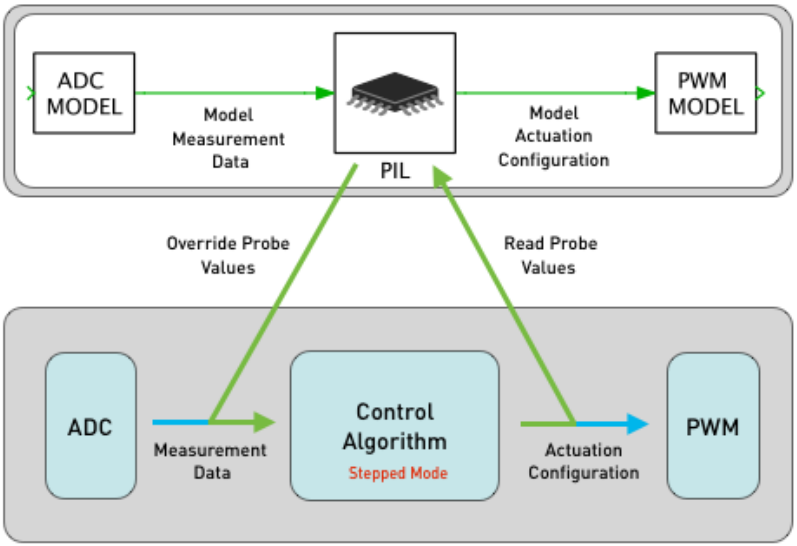
When developing embedded control algorithms, it is quite common to be testing such code, or portions thereof, by executing it inside a circuit simulator. Using PLECS, this can be easily achieved by means of a C-Script or DLL block. This approach is referred to as *Software-in-the-loop* (SIL). A SIL simulation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



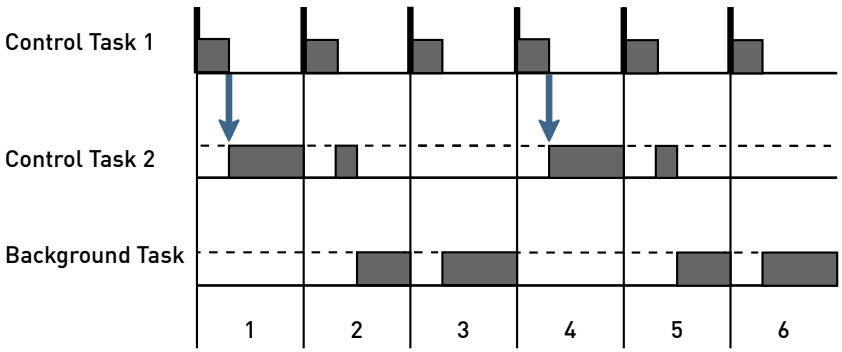
Principle of a PIL simulation

- 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*.

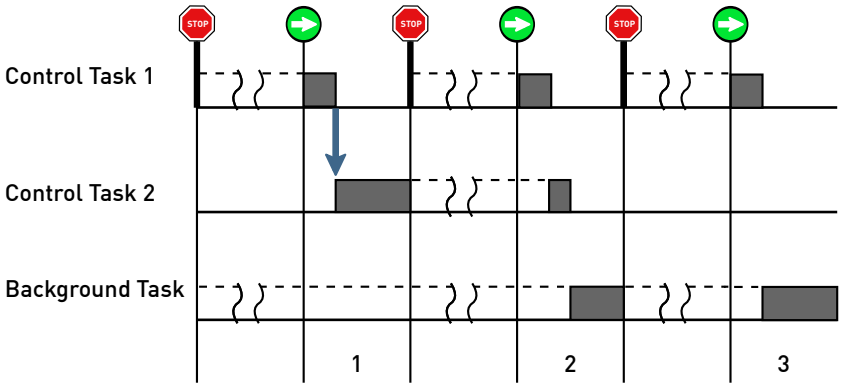
Let us further examine the pseudo real-time operation in the context of an embedded application utilizing nested control loops where fast high-priority tasks (such as current control) interrupt slower lower-priority tasks (such as voltage control). An example of such a configuration with two control 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 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.



Pseudo real-time operation

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 conjunction 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

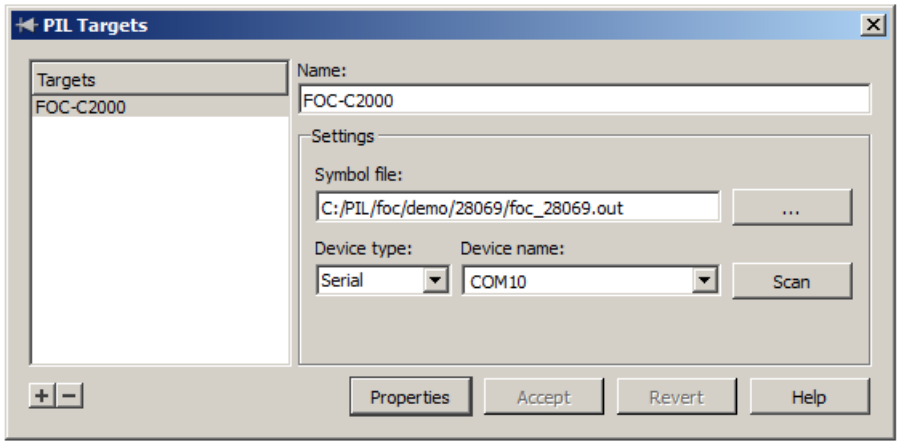
Once an embedded application is equipped with the PIL framework, and appropriate Override Probes and Read Probes are defined, it is ready for PIL simulations with PLECS.

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

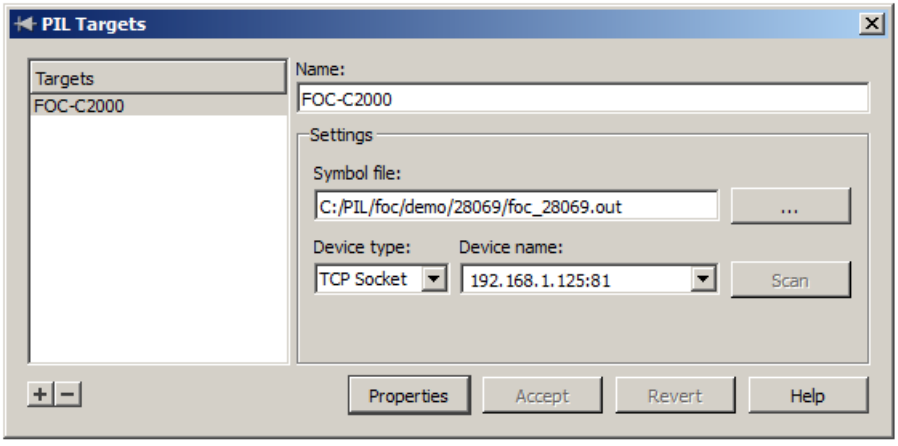
The **Serial device** selection corresponds to conventional physical or virtual serial communication ports. On a Windows machine, such ports are labeled `COMn`, where `n` is the number of the port.

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

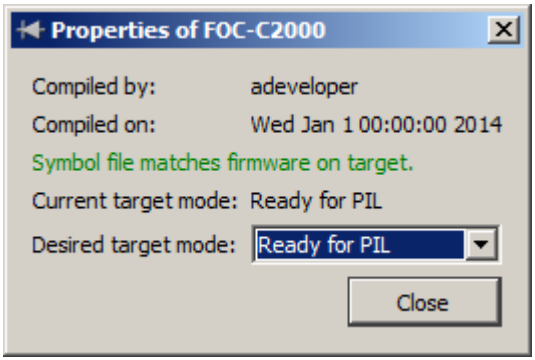
In this case the **Device name** corresponds to the IP address (or URL) and port number, separated by a colon (:).

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.

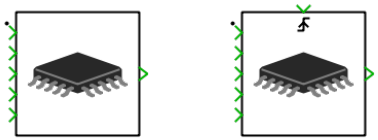


Target Properties

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.

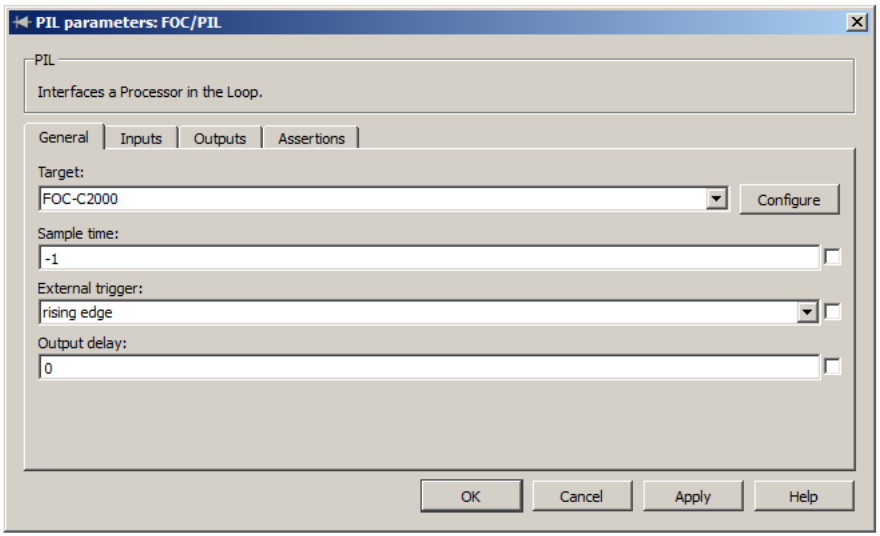
PIL Block

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]**.

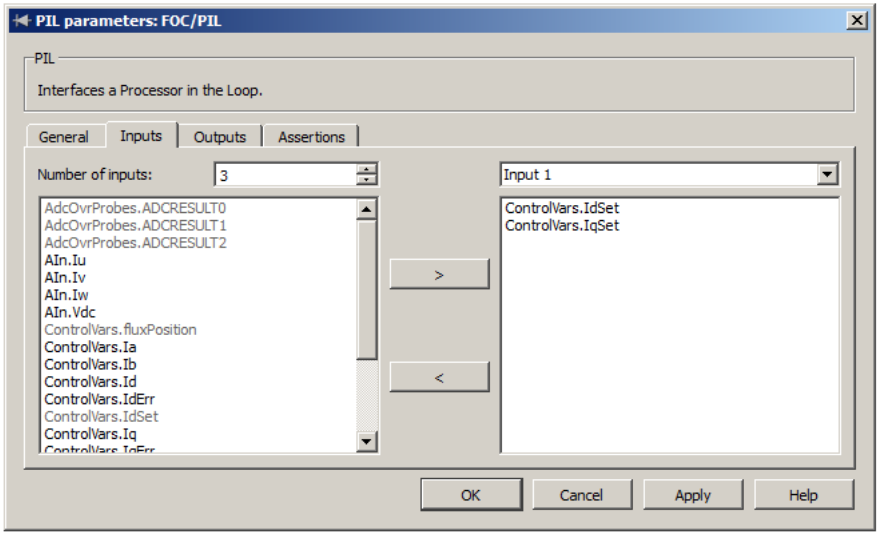
A trigger port can be enabled using the **External trigger** combo box. This is useful if the control interrupt source is part of the PLECS circuit, such as an ADC or PWM peripheral model.

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 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 Override 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 and 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 them up and down the list.

2

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™ MCU family. 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.

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_SYMBOL_DEF` macro. For example, the statement below configures variable `Vdc` for read access by PLECS.

```
uint16_t Vdc;

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`.

```
typedef struct
{
  int q;          //!< fixed-point location
  float ref;      //!< reference value
  char *unit;     //!< unit string
} pil_var;

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 unit-less variable by using a reference value of 1.0 and setting an empty string (“”) for the unit.

The same convention can be used to configure floating point variables as Override Probes or Read Probes.

```
float MotorSpeed;

PIL_SYMBOL_DEF(MotorSpeed, 0, 1.0, "rpm");
```

The first parameter of the `PIL_SYMBOL_DEF` 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 $0.5 * 1800\text{rpm} = 900\text{rpm}$.

```
float MotorSpeed;

PIL_SYMBOL_DEF(MotorSpeed, 0, 1800.0, "rpm");
```

It is also possible to configure structure members, as shown below. Notice how the name of the structure definition is prefixed to the name of the member.

```
struct BATTERY {
  int16_t voltage;
};

struct BATTERY MyBattery;

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

Override Probes

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

```
struct BATTERY {
  PIL_OVERRIDE_PROBE(int16_t, voltage);
};

struct BATTERY MyBattery;
```

The `PIL_OVERRIDE_PROBE` macro expands into the definition of two helper symbols which enable the `MyBattery.voltage` variable to be overriden by PLECS.

```
struct BATTERY {
  int16_t voltage;
  int16_t voltage_probeV;
  int16_t voltage_probeF;
};
```

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 attributes of an Override Probe are configured in the same fashion as Read Probes by the `PIL_SYMBOL_DEF` macro as previously described, e.g.:

```
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 Probes 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.

If an Override Probes 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.

```
Battery.voltage = measureBattVolt();

PLX_VECT_parkRot(ControlVars.Ia, ControlVars.Ib, \
  &ControlVars.Id, &ControlVars.Iq, \
  ControlVars.fluxPosSin, ControlVars.fluxPosCos);
```

Original code without use of Override Probes

Assume that during PIL simulations, we would like to override 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.

```
SET_OPROBE(Battery.voltage, measureBattVolt());

int16_t id, iq;

PLX_VECT_parkRot(ControlVars.Ia, ControlVars.Ib, \
  &id, &iq, \
  ControlVars.fluxPosSin, ControlVars.fluxPosCos);

SET_OPROBE(ControlVars.Id, id);
SET_OPROBE(ControlVars.Iq, iq);
```

Use of Override Probes

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.

Code Identity

PLECS accesses Override Probes and Read Probes by address (as opposed to name). The PIL block extracts the address of those directly 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 read. 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 14), the target manager can be used to verify the match of the selected binary file.

The GUID is generated at compile time by calling the `CIDGen.exe` tool (included with the PIL demo projects). This generates a header file with the following constants:

- `CODE_GUID` – Globally unique identifier
- `COMPILE_TIME_DATE_STR` – Compile time stamp formatted as string
- `COMPILE_TIME_DATE_INT` – Compile time stamp formatted as long integer
- `USER_NAME` – Log-on name of person who compiled the code

`CIDGen.exe` takes one parameter, the name of the header file to be generated. The listing below shows the contents of such a generated header file. Please refer to the Plexim sample projects for more information on how to call the utility from your IDE.

```
#ifndef CID_GEN_H
#define CID_GEN_H

#define CODE_GUID {0xA8,0x45,0x11,0xDE,0x05,0x4C,0xAC,0x41}
#define COMPILE_TIME_DATE_STR "Sun May 30 10:11:43 2010"
#define COMPILE_TIME_DATE_INT 0x4C02721F
#define USER_NAME "john doe"

#endif
```

cid.h File

The value of `CODE_GUID` is passed to the PIL framework during initialization; see "Framework Configuration" (on page 39). The value must also be assigned to the `PIL_D_Guid` constant as follows:

```
const unsigned char PIL_D_Guid[] = CODE_GUID;
```

The other values generated by `CIDGen.exe` can be used for diagnostics purposes using PIL constants, as demonstrated in section "Configuration Constants" (on page 40).

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 39.

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 following 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.

```
void SCIPoll()
{
  while(SciaRegs.SCIFFRX.bit.RXFFST != 0)
  {
    // a character has been received
    PIL_RA_serialIn((int16)SciaRegs.SCIRXBUF.all);
  }

  int16_t ch;
  if(SciaRegs.SCICTL2.bit.TXRDY == 1)
  {
    // link is ready for transmission
    if(PIL_RA_serialOut(&ch))
    {
      SciaRegs.SCITXBUF = ch;
    }
  }
}
```

SCI callback

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.

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.

	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

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.

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

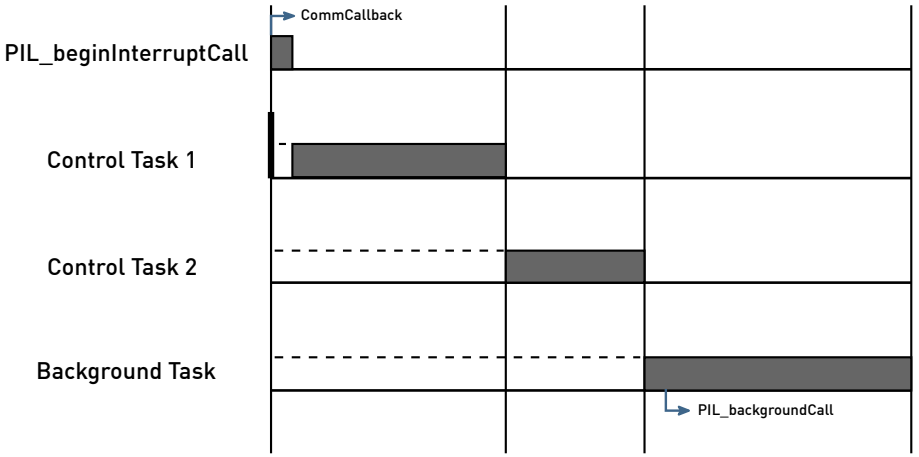
Now let us examine the operation of the framework in both real-time and pseudo real-time mode.

The figure on page 32 shows the framework operation in non-PIL (real-time) mode.

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.

```
/**
 * Main interrupt routine
 */
Void TickFxn(UArg arg)
{
  PIL_beginInterruptCall();

  // fast control task
  ControlTask1();

  // slow control task
  divider++;
  if(divider == TASK2_PERIOD)
  {
    divider = 0;
    Swi_post(Swi);
  }
}

/**
 * Software interrupt for slow control task
 */
Void SwiFxn(UArg arg0, UArg arg1)
{
  ControlTask2();
}

/**
 * Background task
 */
Void BackgroundTaskFxn(Void)
{
  PIL_backgroundCall();
}
```

Control Task Dispatching



PIL framework during real-time operation

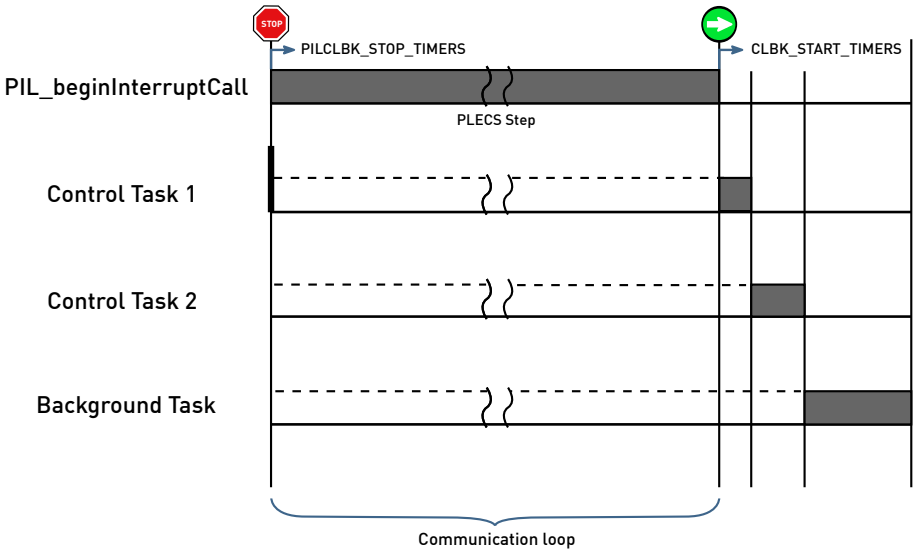
Note During real-time operation, the PIL framework must have a minimal influence on the timing of the dispatched control tasks. Therefore the `CommCallback` function must be implemented as efficiently as possible.

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.

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.



PIL framework during pseudo real-time operation

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:

```
void PilCallback(PIL_CtrlCallbackReq_t aCallbackReq)
{
  switch(aCallbackReq)
  {
    case PIL_CLBK_STOP_TIMERS:
      //application specific code
      break;
    case PIL_CLBK_START_TIMERS:
      //application specific code
      break;
      .
      .
      .
    default:
      //catching an undefined callback
      break;
   }
}
```

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

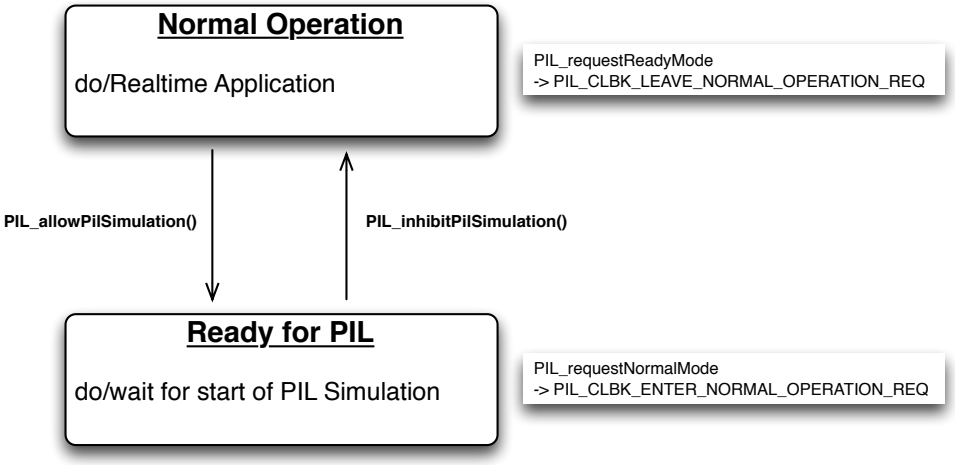
- `PIL_CLBK_ENTER_NORMAL_OPERATION_REQ` – Called when the target mode "Normal Operation" has been requested. The application must indicate that it has entered normal operation by executing `PIL_inhibitPilSimulation()`.
- `PIL_CLBK_LEAVE_NORMAL_OPERATION_REQ` – Called when the target mode "Ready for PIL" has been requested. The application must confirm that it is ready for PIL simulations by executing `PIL_allowPilSimulation()`.
- `PIL_CLBK_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 13) the PIL framework distinguishes between the two target modes.



PIL target modes and mode change requests

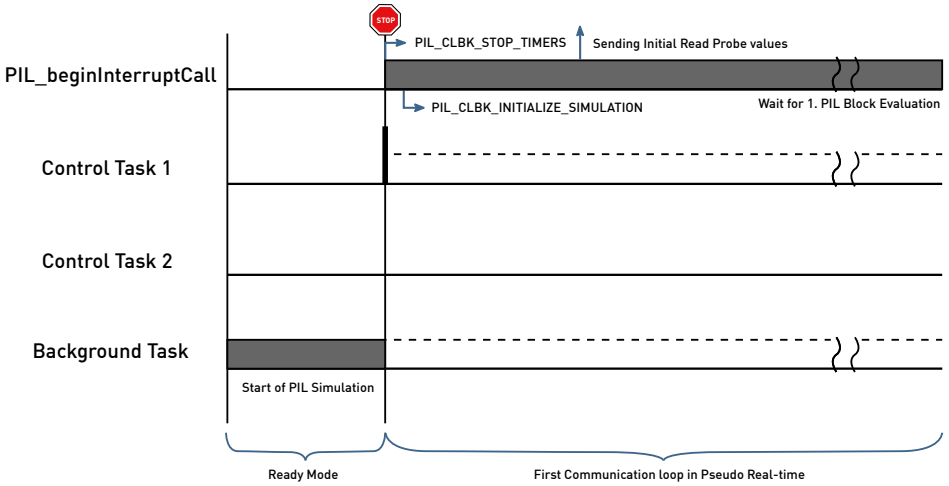
In Normal Operation mode, the target executes in true real-time operation driving the load with an active power stage. PIL simulations are inhibited in this mode due to the power stage being active. A PIL simulation can only be started if the target is in Ready for PIL mode, which corresponds to a safe 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

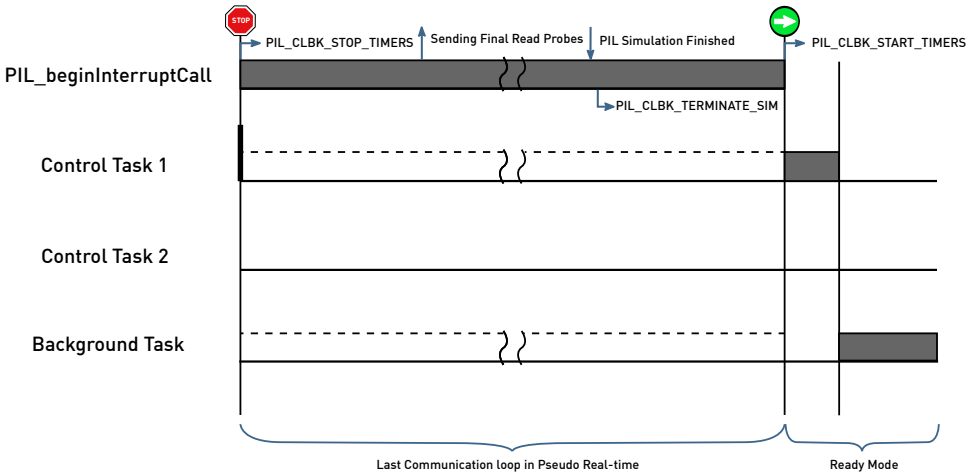
When running multiple PIL simulations and comparing results it is important that all simulation-runs begin with identical initial conditions. This is achieved by means of the `PIL_CLBK_INITIALIZE_SIMULATION` request, which is issued via the control callback at the beginning of a simulation.



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.

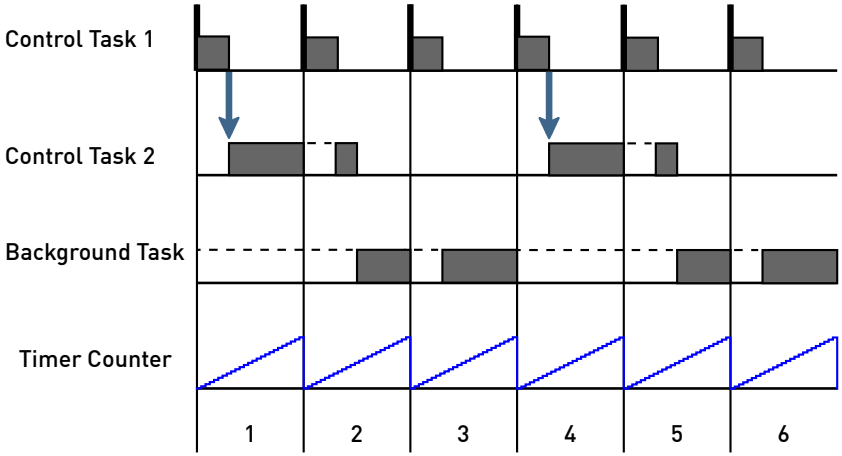
At the end of a PIL simulation, a `PIL_CLBK_TERMINATE_SIMULATION` request is issued prior to returning to real-time operation.



End of a PIL Simulation

Control Dispatching

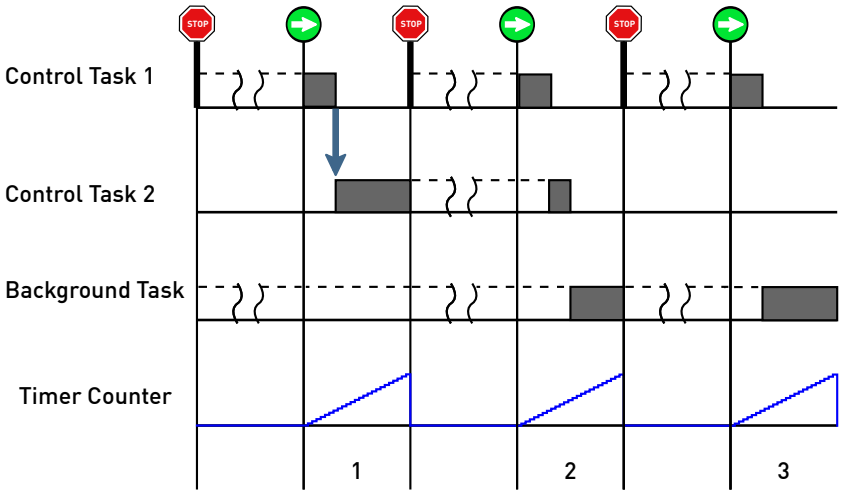
During a PIL simulation, the target operates in a pseudo real-time fashion with the execution of the control tasks being paced and synchronized with the simulation.



Real-time operation with timer

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.

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.



Pseudo real-time operation with periodically stopped timer

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.

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.

```
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;
   }
}
```

Active task synchronization via simulation initialization callback

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

- `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.

Optional configurations are as follows:

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

Configuration Constants

`PIL_D_` symbols of type qualifier `const` are used for making settings and diagnostics information available to PLECS. At a minimum, `PIL_D_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 `PIL_D_BaudRate` definition. Optionally, further constants can be defined as shown below.

```
const unsigned char PIL_D_Guid[] = CODE_GUID;
const uint32_t PIL_D_BaudRate = BAUD_RATE;

// for diagnostic purposes
const unsigned char PIL_D_CompiledDate[] = COMPILE_TIME_DATE_STR;
const unsigned char PIL_D_CompiledBy[] = USER_NAME;
const char PIL_D_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 to be removed from the binary file.

Initialization Constants

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

`PIL_C_` symbols are used for defining such constants. They must be of `const` integer or float type. Strings and arrays are not supported.

```
const uint32_t PIL_C_SysClk = SYSCLK_HZ;
const uint32_t PIL_C_PwmFrequency = PWM_HZ;
const uint32_t PIL_C_ControlFrequency = CONTROL_HZ;
const uint16_t PIL_C_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;
```

3

TI C2000 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.

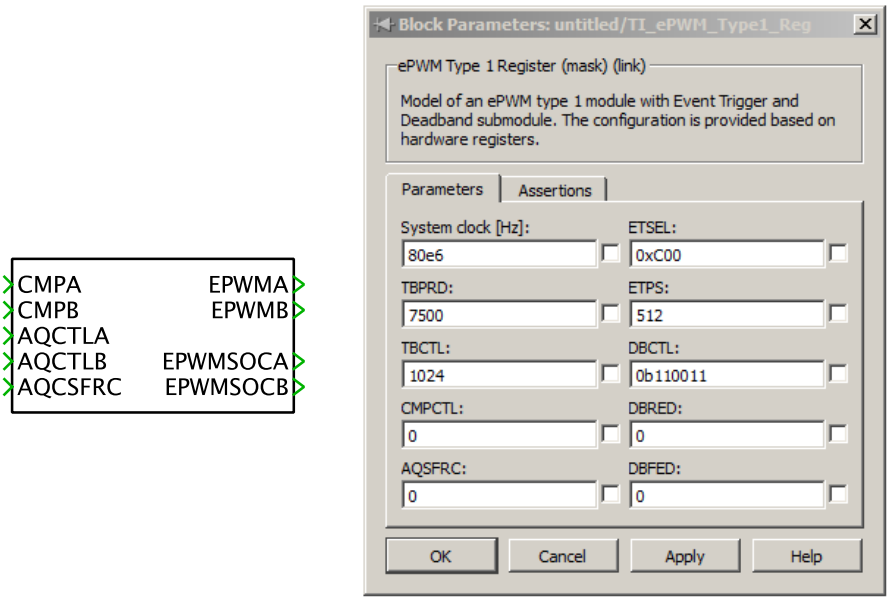
The PLECS PIL library includes high-fidelity MCU peripheral models which work at the register level, and are therefore well-suited for PIL simulations. Furthermore, certain blocks have a second implementation with a graphical user interface (GUI) that automatically determines the register configurations 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

Enhanced Pulse Width Modulator (ePWM) Type 1

The PLECS peripheral library provides two blocks for the TI ePWM type 0/1 module. One block has a register-based configuration mask and a second block features a graphical user interface. In both cases, you should distinguish between registers configured in the parameter mask and inputs to the block. Mask parameters are fixed (static) during simulation and correspond to the configurations which the embedded software makes during the initialization phase. Inputs are dynamically changeable while the simulation is running. The fixed configuration can be entered either using a register-based approach or a graphical user interface, while the dynamic values supplied at the inputs must correspond to raw register values. The figure below shows the block and its parameters for the register-based version.

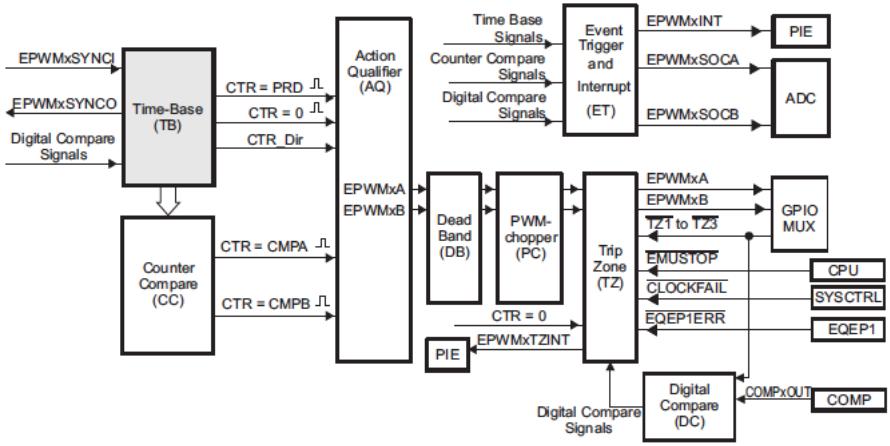


Register based ePWM module model

As depicted above, the block can be configured directly using the registers of the hardware module, making it possible to exactly mirror the configuration applied to the target. Also as shown, either hexadecimal, decimal or binary representation can be used to enter the configuration.

Supported Submodules and Functionalities

The ePWM type 0/1 module consists of several submodules:



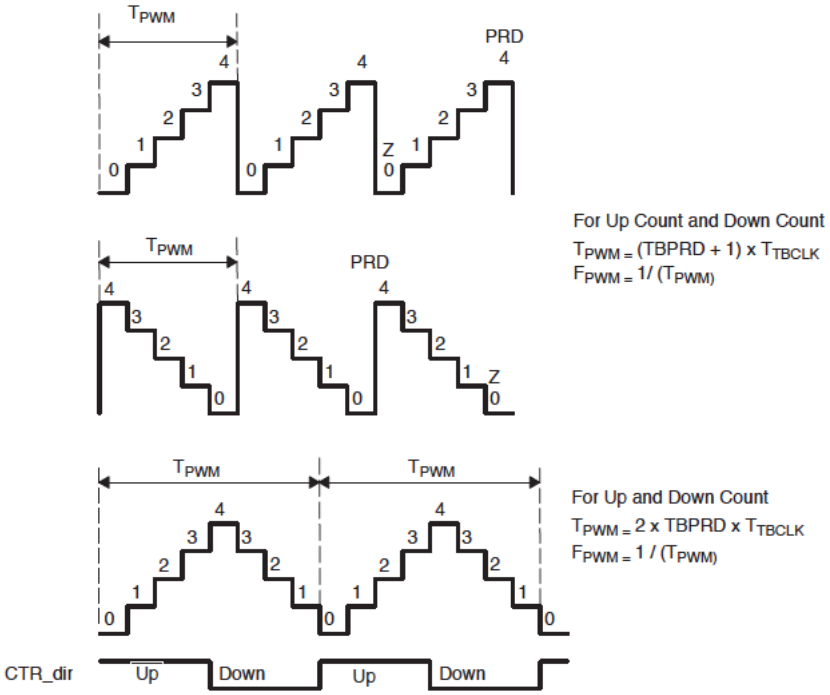
Submodules of the ePWM type 1 module [1]

The PLECS ePWM model accurately reflects the most relevant features of the following submodules:

- Time-Base submodule
- Counter-Compare submodule
- Action-Qualifier submodule
- Dead-Band submodule
- Event-Trigger submodule

Time-Base (TB) Submodule

This submodule realizes a counter that can operate in three different modes for the generation of asymmetrical and symmetrical PWM signals. The three modes, *up-count*, *down-count*, and *up-down-count*, are visualized below.



Counter modes and resulting PWM frequencies [1]

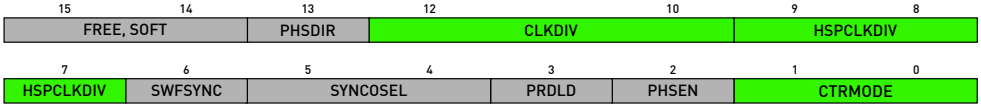
In *up-count mode*, the counter is incremented from 0 to a counter period *TBPRD* using a counter clock with period T_{TBCLK}. When the counter reaches the period, the subsequent count value is reset to zero and the sequence is repeated. When the counter is equal to zero or the period value, the submodule produces a pulse of one counter clock period, which, together with the actual counter direction, is sent to the subsequent Action Qualifier submodule.

The period of the timer clock can be calculated based on the system clock (*SYSCLKOUT*) and the two clock dividers (*CLKDIV* and *HSPCLKDIV*) by:

$$T_{TBCLK} = \frac{CLKDIV \cdot HSPCLKDIV}{SYSCLKOUT}$$

The resulting PWM period further depends on the counting mode, the counter period (*TBPRD*) and the counter clock period as depicted in the figure above.

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 *TBCTL* register.



TBCTL Register Configuration [1]

The *CLKDIV* and *HSPCLKDIV* cells select the desired clock dividers and the *CTRMODE* cell defines the counter mode. Only counter modes 00, 01, and 10 are supported by the PLECS ePWM model.

Example Configuration – Step 1

This example is based on the parameter mask shown at the beginning of this chapter and will be further developed in subsequent sections. The *TBCTL* register is configured to:

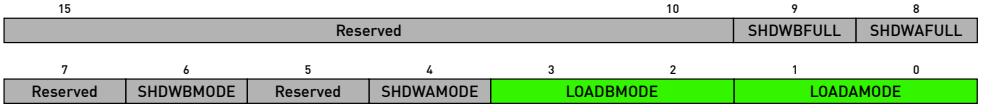
$$TBCTL = 1024 \quad \hat{=} \quad 0\,0\,0 \;\underbrace{0\,0\,1}_{CLKDIV} \;\underbrace{0\,0}_{HSPCLKDIV} \quad 0\,0\,0\,0\,0\,0 \;\underbrace{0\,0}_{CTRMODE}$$

According to this configuration, the time base submodule is operating in the *up-count* mode with a timer clock period twice the system-clock period. The resulting PWM signal has the following period:

$$T_{PWM} = (TBPRD + 1) \cdot \frac{CLKDIV \cdot HSPCLKDIV}{SYSCLKOUT} = 187.525\ \mu s.$$

Counter-Compare (CC) Submodule

This submodule is responsible for generating the pulses *CTR* = *CMPA* and *CTR* = *CMPB* used by the Action-Qualifier submodule. In a typical application, the compare values change continuously during operation and therefore need to be part of the dynamic configuration (block inputs). The PLECS implementation only supports the shadow mode for the *CMPx* registers, i.e. the content of a *CMPx* register is only transferred to the internal configuration at reload events. The reload events are specified in the *CMPCTL* register.



CMPCTL Register Configuration [1]

For efficiency, the PLECS ePWM model only supports the following combinations of counter mode and reload events:

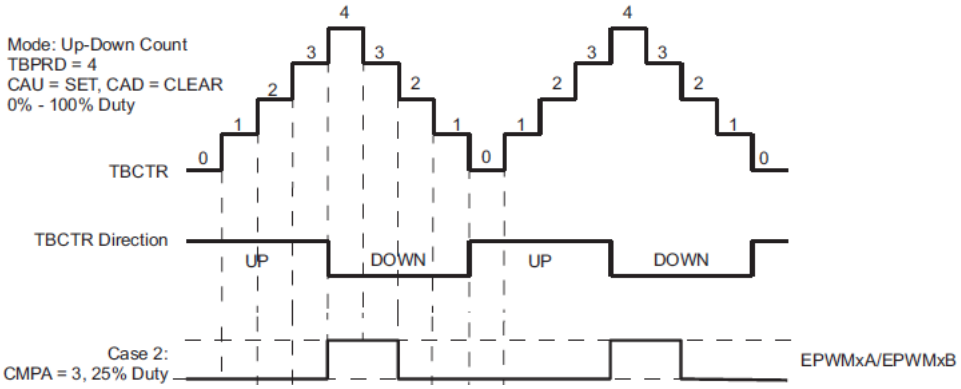
CTRMODE	LOADAMODE	LOADBMODE
Up-count	CTR = 0	CTR = 0
Down-count	CTR = PRD	CTR = PRD
Up-down-count	CTR = 0 or CTR = 0 or CTR = PRD	CTR = 0 or CTR = 0 or CTR = PRD

Furthermore, only coinciding configurations for *LOADAMODE* and *LOADBMODE* are supported.

In the example configuration, the *CMPCTL* register needs to be set to 0 because the counter is operating in up-count mode.

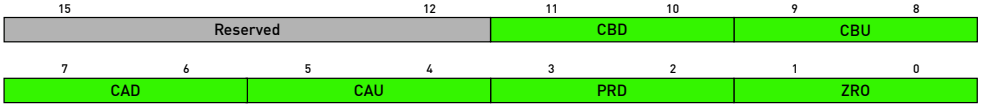
Action-Qualifier (AQ) Submodule

This submodule sets the *EPWMx* outputs based on the flags generated by the Time-Base and Counter-Compare submodules. The *AQCTLx* registers configure the actions to be performed at the different events. Similiar to the *CMPx* registers, the *AQCTLx* registers are operated in shadow mode and are reloaded at both the zero and the period event.



ePWM timing example [1]

The figure above shows an example (Case 2) where the *ePWM* output is set to high at the *CTR = CMPA* event. As depicted, an output change always lags the event by one counter clock period. The following shows the structure of the *AQCTL* register.



AQCTL Register Configuration [1]

Actions depend on the counter direction. For example, the register cell *CBD* defines what happens to the corresponding *ePWMx* output when the counter equals *CMPB*, when the counter is counting down. The following configurations exist:

- 00 - No Action
- 01 - Force ePWMx output low
- 10 - Force ePWMx output high

- 11 - Toggle ePWMx output

If events occur simultaneously, the *ePWM* module respects a priority assignment based on the counter mode. The following figures show the Action-Qualifier prioritization.

Priority Level	Event If TBCTR is Incrementing TBCTR = Zero up to TBCTR = TBPRD	Event If TBCTR is Decrementing TBCTR = TBPRD down to TBCTR = 1
1 (Highest)	Software forced event	Software forced event
2	Counter equals CMPB on up-count (CBU)	Counter equals CMPB on down-count (CBD)
3	Counter equals CMPA on up-count (CAU)	Counter equals CMPA on down-count (CAD)
4	Counter equals zero	Counter equals period (TBPRD)
5	Counter equals CMPB on down-count (CBD)	Counter equals CMPB on up-count (CBU)
6 (Lowest)	Counter equals CMPA on down-count (CAD)	Counter equals CMPA on up-count (CBU)

Action-Qualifier prioritization in up-down-count mode [1]

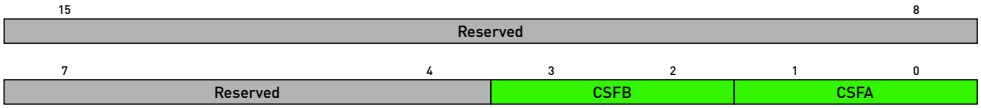
Priority Level	Event
1 (Highest)	Software forced event
2	Counter equal to period (TBPRD)
3	Counter equal to CMPB on up-count (CBU)
4	Counter equal to CMPA on up-count (CAU)
5 (Lowest)	Counter equal to Zero

Action-Qualifier prioritization in up-count mode [1]

Priority Level	Event
1 (Highest)	Software forced event
2	Counter equal to Zero
3	Counter equal to CMPB on down-count (CBD)
4	Counter equal to CMPA on down-count (CAD)
5 (Lowest)	Counter equal to period (TBPRD)

Action-Qualifier prioritization in down-count mode [1]

Notice how software-forced events have the highest priority in all three count modes. Software forcing is configured by the Action-Qualifier-Continous-Software-Force-Register (*AQCSFRC*), provided as an input to the PLECS block to allow dynamic register configuration.

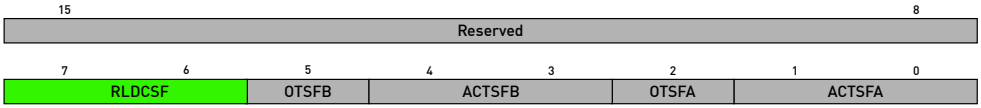


AQCSFRC Register Configuration [1]

The figure above shows the relevant cells of the register where *CSFA* and *CSFB* can be used to force an output. The following configurations are supported:

- 00 - Forcing Disabled
- 01 - Force a continuous low on ePWMx
- 10 - Force a continuous high on ePWMx
- 11 - Forcing Disabled

As illustrated in the previous ePWM timing example, the change of an ePWMx output lags the change of *AQCSFRC* by one counter clock period. Similar to the previously described registers with dynamic configuration, the *AQCSFRC* register is operated in shadow mode. The reload events can be defined with the *AQSFRC* register.



AQSFRC Register Configuration [1]

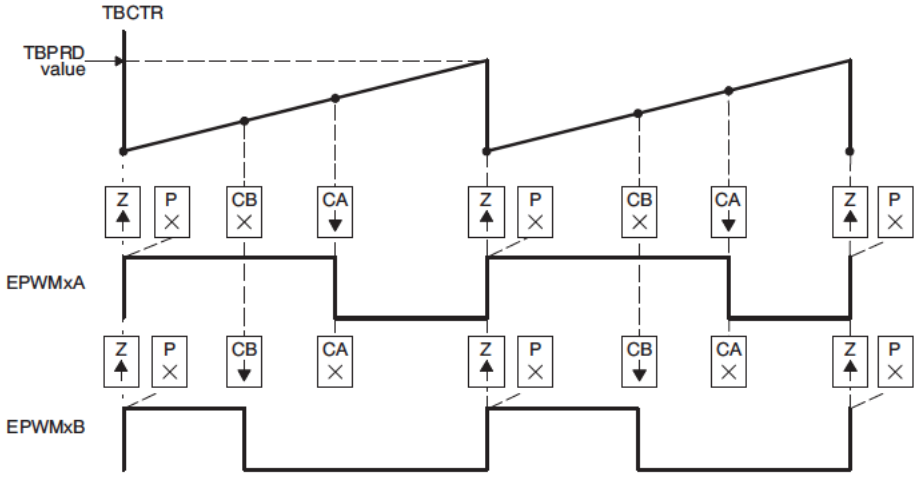
The supported modes for *RLDCSF* are listed below.

- 00 - CTR = Zero
- 01 - CTR = PRD
- 10 - CTR = Zero or CTR = PRD

Immediate mode for loading is not supported due to implementation efficiency reasons.

Example Configuration – Step 2

The following figure shows an example using the actions defined by the *AQCTL* registers. Refer to [1] for a detailed explanation of the action symbols.



Desired ePWMA and ePWMB output signals [1]

To realize the above *ePWM* signals, the dynamic configuration must be set as follows:

$$CMPA = 3500, CMPB = 2000, AQCSFRC = 0$$

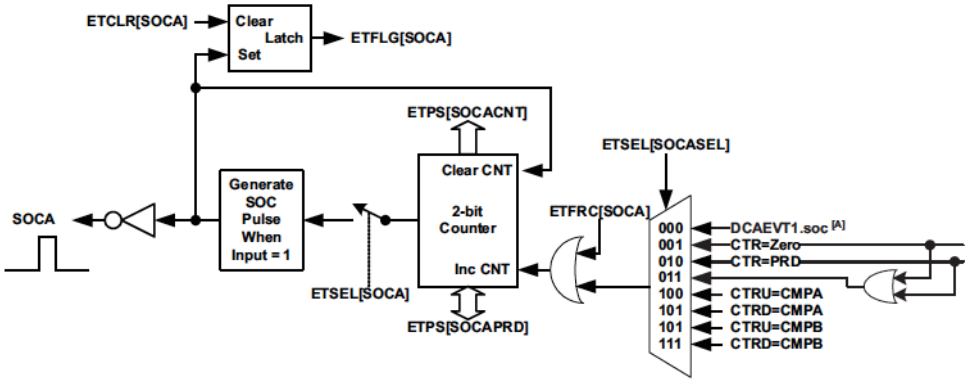
Furthermore, the Action-Qualifier must be set as shown below:

$$AQCTLA = 18 \quad \hat{=} \quad 0\,0\,0\,0 \; \underbrace{0\,0}_{CBD}\,\underbrace{0\,0}_{CBU}\;\underbrace{0\,0}_{CAD}\,\underbrace{0\,1}_{CAU}\;\underbrace{0\,0}_{PRD}\,\underbrace{1\,0}_{ZRO}$$

$$AQCTLB = 258 \quad \hat{=} \quad 0\,0\,0\,0 \; \underbrace{0\,0}_{CBD}\,\underbrace{0\,1}_{CBU}\;\underbrace{0\,0}_{CAD}\,\underbrace{0\,0}_{CAU}\;\underbrace{0\,0}_{PRD}\,\underbrace{1\,0}_{ZRO}$$

Event-Trigger (ET) Submodule

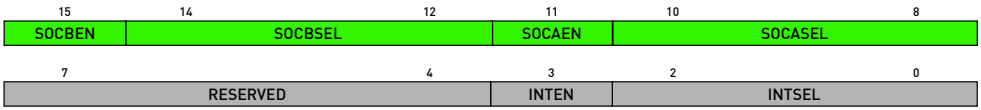
This submodule utilizes the signals generated by the Time Base and Counter Compare submodules to generate events (pulses) at the *ePWMSOCx* outputs. Such pulses can trigger an ADC conversion or invoke the execution of a control algorithm or PIL block. For each *ePWMSOC* channel, the Event Trigger module provides an internal 2-bit counter which permits a downsampling of events. The following diagram shows the internal structure for the example of *SOCA*.



Event Trigger Logic [1]

As can be seen, the counter is being incremented using one of the source signals on the right-hand side. The incrementing source signal is selected by the *SOCxSEL* field. An *SOC* pulse is generated when the *SOCxCNT* reaches its configurable period (*SOCxPRD*) and pulse generation is activated by the *SOCx* flag. The configuration for both the *SOCA* and *SOCB* portion of the Event Trigger is set by the registers *ETSEL* and *ETPS*, which are realized as static parameters of the PLECS model.

The *ETSEL* register has the following structure.

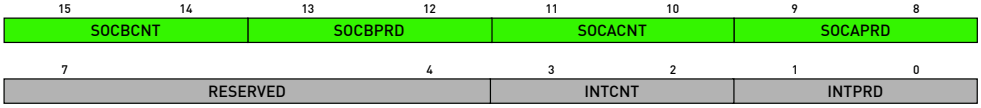


ETSEL Register Configuration [1]

The *SOCxEN* bits activate or deactivate the *SOCx* pulses. The *SOCxSEL* cells

determine the source for the event trigger counter. Note, $SOCxSEL = 000$ is not supported in the model.

This figure shows the structure of the *ETPS* register.



ETPS Register Configuration [1]

The *SOCxCNT* cells allow initialization of the event counter. The *SOCxPRD* bits determine the number of events that must occur before an *SOCx* pulse is generated. Refer to [1] for detailed information regarding the configuration of the *ETPS* register.

Example Configuration – Step 3

A possible use case for the Event-Trigger submodule is to generate a *SOCA* pulse every second time the TB-counter meets the *CMPA* value. To achieve this behavior, the ET is configured as follows.

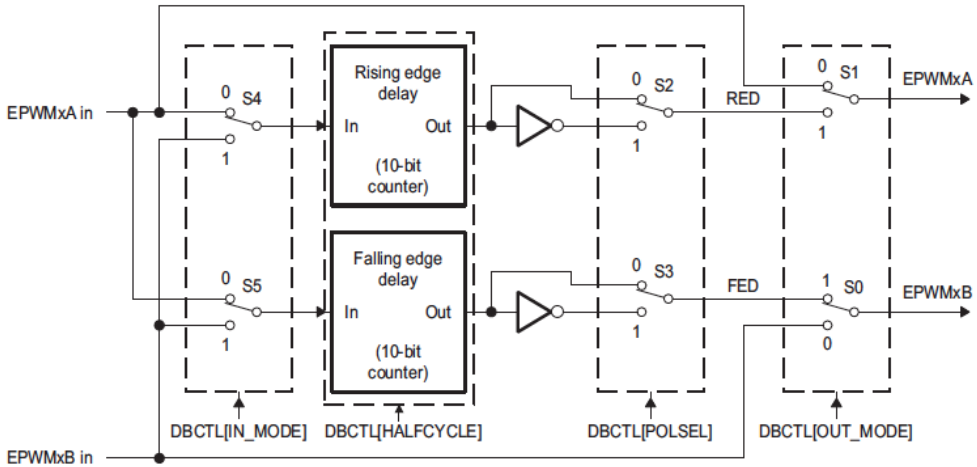
$$ETSEL = 0xC00 \quad \hat{=} \quad 0000 \quad \underbrace{1}_{SOCAEN} \quad \underbrace{100}_{SOCASEL} \quad 0000 \quad 0000$$

This setting enables the *SOCA* pulses and uses the *CTR = CMPA* event for incrementing the ET-counter. Note that *SOCB* pulses are completely disabled in this example.

$$ETPS = 512 \quad \hat{=} \quad 0000 \quad \underbrace{00}_{SOCACNT} \quad \underbrace{10}_{SOCAPRD} \quad 0000 \quad 0000$$

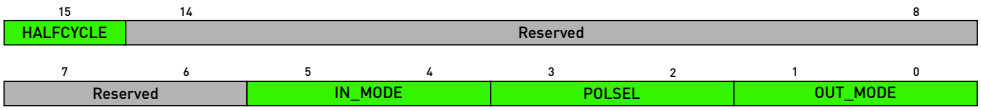
Dead-Band Submodule

The role of this submodule is to add programmable delays to rising and falling edges of the *ePWM* signals and to generate signal pairs with configurable polarity. The figure below depicts the internal structure of the Dead-Band submodule.



Dead-Band Logic [1]

As shown, the *PWMx* signals from the Action-Qualifier submodule are post-processed based on the *DBCTL* register settings. Furthermore, the delay times are programmables by the registers *DBRED* and *DBFED* for the rising and falling edge delay, respectively. The structure of the *DBCTL* register is shown in the following block diagram.



DBCTL Register Configuration [1]

The submodule register cells allow for the following settings:

- *HALFCYCLE* - Delay counters increment with half TB-counter clock period
- *IN_MODE* - Choose source for delay counters; can also be used for output switching

- *POL_SEL* - Invert output polarity
- *OUT_MODE* - Enables Dead-Band bypassing for both outputs

Refer to [1] for detailed information regarding the configuration of the *DBCTL* register.

Example Configuration – Step 4

In the sample configuration, the signal *EPWMB* is selected as the source for both delay counters. Further, both the rising and falling edge of the outputs are delayed by 10 counter clock periods and the polarities are not inverted. The *DBCTL* register therefore should be configured as follows.

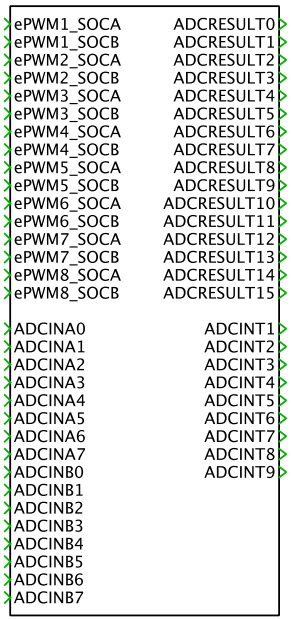
$$DBCTL = 0b110011 \;\hat{=}\; 0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \quad \underbrace{1\,1}_{IN\_MODE} \quad \underbrace{0\,0}_{POL\_SEL} \quad \underbrace{1\,1}_{OUT\_MODE}$$

With the *HALFCYCLE* bit set to zero, the *DBRED* and *DBFED* must be configured to:

$$DBRED = 10 \;\;,\;\; DBFED = 10$$

Analog Digital Converter (ADC) Type 3

The PLECS peripheral library provides two blocks for the TI ADC type 3 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 ADC module model

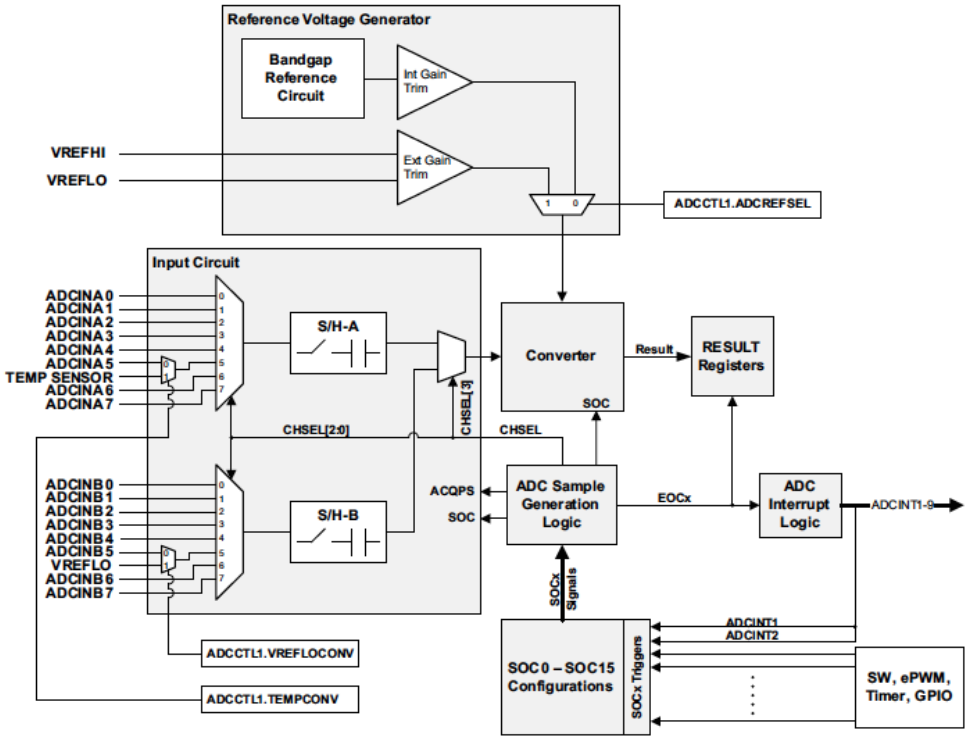
The register-based version allows the user to directly enter register values in decimal, binary, or hexadecimal notation. For convenience, the peripheral library also provides a component with a graphical user interface to simplify the configuration.

Both ADC blocks interface with other PLECS components over the following terminal groups.

- *ePWMx_SOCy* - input ports to trigger ADC conversions
- *ADCINA/B* - input ports for measurements
- *ADCRESULTx* - output ports to access conversion results
- *ADCINTx* - output ports for subsequent logic triggered by a conversion end

ADC Module Overview

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



Overview of the type 3 ADC module [1]

The ADC model implements these logical submodules:

- ADC Converter with result registers
- ADC Reference Voltage Generator
- ADC Sample Generation Logic
- ADC Input Circuit
- ADC Interrupt Logic

ADC Converter with result registers

The type 3 ADC module contains a single 12-bit converter. Either an internal or an external voltage reference can be selected.

The converter takes 13 ADC clocks for a single conversion. The period of an ADC clock, and therefore the time base for the module, is determined based on the system clock and the two clock dividers specified in the *ADCCTL2* register.

ADCCTL2 Register structure [1]

By using the bits *CLKDIV4EN* and *CLKDIV2EN* the ADC time base can be specified as follows.

CLKDIV2EN	CLKDIV4EN	ADC clock
0	0	SYSCLK
0	1	SYSCLK
1	0	SYSCLK / 2
1	1	SYSCLK / 4

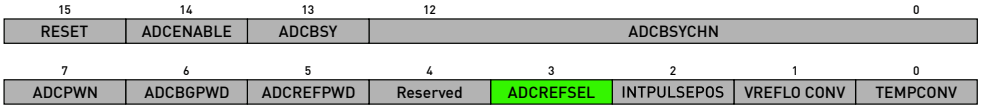
The bit *ADCNONOVERLAP* determines if an overlap of sampling and conversion is allowed in case of multiple pending conversion requests.

- 0 - Overlap is allowed
- 1 - Overlap is not allowed

Once a conversion has completed, the result is stored to one of the 16 result registers *ADCRESULT0 - ADCRESULT15*. These are directly associated with the *SOC*. The content of the result registers is available at the output ports of the model. The representation of the conversion result can be chosen with the mask parameter **Output Mode**.

ADC Reference Voltage Generator

The ADC can use an internal or an external reference voltage. The internal bandgap range is *[0V...3.3V]*, while the external reference can be specified in the component mask.



ADCCTL1 Register structure [1]

With the bit *ADCREFSEL*, the desired voltage reference can be chosen.

- 0 - Internal bandgap
- 1 - Reference voltages defined by module mask

The component only supports the late interrupt pulse mode. Therefore the bit *INTPULSEPOS* should be one.

ADC Sample Generation Logic

The ADC Sample Generation Logic responds to the *SOCx* signals, which are based on 16 individual sets of configuration parameters *SOC0 - SOC15*. Every *SOC* contains the following information:

- Size of Sampling Window (*ACQPS*)
- Converted Input Channel (*CHSEL*)
- Trigger Signal (*TRIGSEL*)

The register used for configuring a *SOC* is shown below.

ADCSOCxCTL Register structure [1]

The register cell *ACQPS* defines the length of the sampling window. The minimum value valid is 06_h which sets the Sample Window to 6+1 ADC clock cycles. Note according to the hardware documentation, there are a number of invalid settings for this register field:

$$10_h\ ,\ 11_h\ ,\ 12_h\ ,\ 13_h\ ,\ 14_h\ ,\ 1D_h\ ,\ 1E_h\ ,\ 1F_h\ ,\ 20_h\ ,\ 21_h\ ,\ 2A_h\ ,\ 2B_h\ ,\ 2C_h$$

$$2D_h\ ,\ 2E_h\ ,\ 37_h\ ,\ 38_h\ ,\ 39_h\ ,\ 3A_h\ ,\ 3B_h$$

The time needed for a full conversion can be calculated with the following equation.

$$T_{conv} = \underbrace{(ACQPS + 1) \cdot ADC_{clk}}_{Sampling Window} + \underbrace{13 \cdot ADC_{clk}}_{Conversion}$$

The *CHSEL* field associates an input pin with a specific *SOC*. The component allows single and simultaneous sampling – see section "ADC Input Circuit" (on page 64). For an *SOC* in single sample mode, cell configuration is as follows.

CHSEL	Input
0h	ADCINA0
1h	ADCINA1
...	...
7h	ADCINA7
8h	ADCINB0
...	...
Fh	ADCINB7

In case of simultaneous sample mode, the channel selection is configured as pairs.

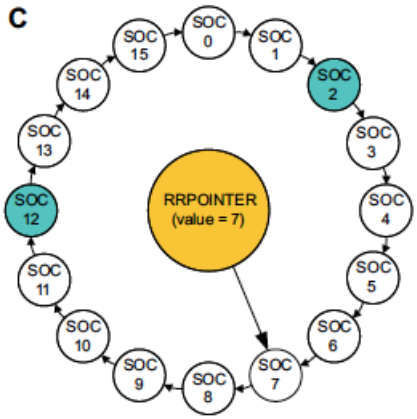
CHSEL	Input pair
0h	ADCINA0 / ADCINB0
1h	ADCINA1 / ADCINB0
...	...
7h	ADCINA7 / ADCINB7
> 7h	Invalid Selection

With the *TRIGSEL* field it is possible to choose a particular trigger source available as a block input. The PLECS component only supports *ePWMx_SOCy* trigger signals. The following table shows the mapping to the hexadecimal representation. Configurations above 14_h and below 05_h are invalid and result in an error.

Additionally, it is possible to configure the interrupt signals *INT1* and *INT2* to trigger ADC conversions. See section "ADC Interrupt Logic" (on page 65) for further details.

TRIGSEL	Input / Source
05h	ePWM1_SOCA
06h	ePWM1_SOCB
07h	ePWM2_SOCA
...	...
14h	ePWM8_SOCB

During operation of an ADC, more than one conversion trigger can occur simultaneously. An *SOC* can also be triggered while a conversion is already active. A round robin method prioritizes pending *SOCs*. This scheme is accurately reflected by the PLECS component. The figure below shows an example snapshot of the round robin wheel.



ADC Prioritization example [1]

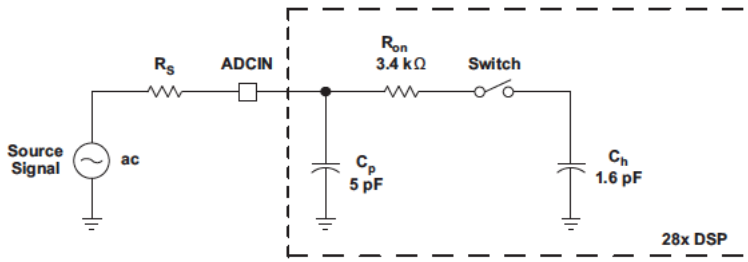
This wheel consist of 16 *SOC* flags and a round robin pointer (*RRPOINTER*). An *SOC* flag is set when a trigger is received and is cleared when the corresponding conversion finishes. The round robin pointer always points to the last converted *SOC* and is changed with the end of every conversion. In the PLECS ADC model, the round robin pointer initially points to *SOC15*. In the example above, the round robin pointer points to *SOC7* indicating this is the last converted *SOC*. At this point in time, the *SOC2* and *SOC12* are triggered and the corresponding flags are set. For prioritization, the ADC starts with

RRPOINTER+1 and goes clockwise through the round robin wheel, meaning *SOC12* is executed next in this example.

The hardware ADC also provides higher prioritized *SOCs* and a *ONESHOT* single conversion mode. These are not supported by the PLECS model.

ADC Input Circuit

The Input Circuit of the type 3 ADC module consists of two separate Sample&Hold circuits (S&H), each connected to a multiplexer. The field *CHSEL* from the *ADCSOCxCTL* register associates an input with a particular *SOC*. Measurements of *TEMP SENSOR* and *VREFLO* are not supported by the PLECS model. The figure below shows the hardware circuit schematic of an *ADCIN* voltage connected to an S&H circuit.

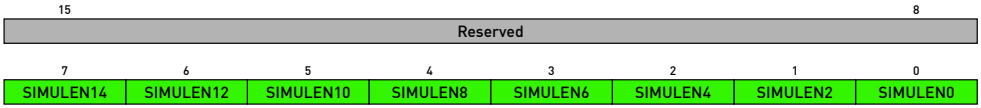


ADCInx Input Model [1]

After an *SOC* is triggered from the round robin wheel, the switch is closed for the sampling window changing the voltage of the sampling Capacitor C_h. Once the sampling time has elapsed, the switch is opened and the conversion starts. For simulation efficiency reasons, the PLECS model of the ADC approximates this behavior by taking the average of the input values at the begin and end of the sampling window.

The type 3 ADC further provides single as well as simultaneous measurements. For a single measurement, only one S&H circuit is active at a time. For simultaneous measurements, both S&H circuits operate in parallel, sampling two different voltages at the same time. The conversion is carried out sequentially starting with the upper S&H voltage. The sampling mode is assigned pairwise, always in groups of even and odd *SOCs* using the register shown below.

With the bit *SIMULENx*, the sampling mode can be chosen as follows.



ADCSAMPLEMODE Register structure [1]

- 0 - Single sample mode for *SOCx* and *SOCx+1*
- 1 - Simultaneous sample mode set for *SOCx* and *SOCx+1*

In case of simultaneous mode, both *SOCs* can still be configured independently by the *ADCSOCxCTL* registers. The behavior during conversion (sample window length and channel selection) is always determined by the triggered *SOC*. For a more advanced understanding of the modules behavior and configuration, please refer to [1].

ADC Interrupt Logic

For every conversion, the ADC sample generation logic generates an end of conversion pulse (*EOC*) with duration one ADC clock period. This pulse is generated one cycle before latching the conversion result. The interrupt pulse always lags the *EOC* pulse by one ADC clock period and therefore is simultaneous to the result latch. The ADC Interrupt Logic can generate the interrupts *ADCINT1-ADCINT9*, which are available at the output ports of the ADC model. With the register below, the interrupt behavior can be configured.

INTSELxNy Register structure for the example of INT1 and INT2 [1]

The *INTxE* bit enables the interrupt generation by an *EOC* flag.

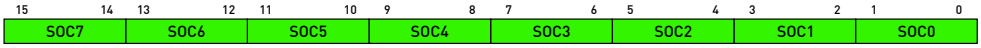
- 0 - *ADCINTx* disabled
- 1 - *ADCINTx* enabled

The *INTxSEL* cell defines which *EOC* flag triggers the interrupt.

INTxSEL	Interrupt Trigger
00h	EOC0 triggers interrupt ADCINTx
01h	EOC1 triggers interrupt ADCINTx
...	...
0Fh	EOC15 triggers interrupt ADCINTx
> 0Fh	Invalid Selection

Note The cells *INT10E* and *INT10SEL* in *INTSEL9N10* have no effect because the model only supports the interrupts *ADCINT1-ADCINT9*.

Additionally, the interrupts *INT1* and *INT2* can be configured to internally trigger *SOCs*, using the the following registers:



ADCINTSOCSEL1 Register structure [1]

ADCINTSOCSEL2 Register structure [1]

The field *SOCx* can be configured as follows.

SOCx	Interrupt Trigger
00	No ADCINT will trigger SOCx
01	ADCINT1 will trigger SOCx
10	ADCINT2 will trigger SOCx
11	Invalid Selection

The setting in this register, if not 00, overwrites the trigger setting defined in the field *TRIGSEL* of the *ADCSOCCTLx* register.

Reference

Texas Instruments: *TMS320x2806x Piccolo Technical Reference Manual*, Literature Number SPRUH18D, January 2011-February 2013

4

Embedded Application

This chapter provides additional information about the C2000 FOC demo application.

Importing the CCS Demo Project

The source code of the embedded demonstration project is provided as part of the PIL Framework installation and can be directly imported into CCSv5.5 by the **Import Existing CCS Eclipse Project** item from the **Project Menu**.

Configuring the Project

The building of the demo project is configured to include custom *pre-build* and *post-build* actions.

At the start of a build, the `CIDGen.exe` utility (included with the project) is called to generate a globally unique identifier (GUID) that can be linked with the code, as explained in “Code Identity” (on page 26). At the completion of the build `C2Prog` is called to generate an extended-hex (*.ehx) file for reflashing the MCU using C2Prog.

These additional build steps are configured in the **Build** section of the project properties (**Steps** tab). Both build steps call the batch file named `buildsteps.bat`.

If you have installed `C2Prog` in a non-default location, you will have to open the buildsteps.bat file with a text editor and adjust the `C2PROG_PATH` setting to match the installation on your machine.

Rebuilding the Project

The project can be compiled and flashed by clicking the “bug” symbol on the toolbar or selecting **Debug** from the **Project** menu.

After reflashing the C2000 MCU with your own project, ensure the PLECS target manager is pointing to the correct symbol file (located in the `Debug` folder).

Project Structure

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

Device Support

Standard files from TI’s ControlSUITE are used for basic device support. The associated header files are located in the `include` folder. Device support C and assembly files are located at the root of the project.

- `DevInit.c`
- `CodeStartBranch.asm`
- `GlobalVariableDefs.c`
- `usDelay.asm`

Linker Files

The principal linker command file `F28xx.cmd` is configured to link the program data to Flash memory, and also defines a special `MEMORY` range and `SECTION` for the probe definitions.

```
MEMORY
{
PAGE 0 :
  ...
PAGE 1:
  ...
PAGE 2 :
  VAR_INFO  : origin = 0x000000, length = 0x1000
}

SECTIONS
{
  ...

    /* Link PIL probe definitions to virtual memory */
    .csconf:       load = VAR_INFO, PAGE = 2,
             TYPE = COPY { pil_symbols.obj (.econst) }

}
```

Notice how a virtual page "2" is used to store the probe symbol information. This makes the symbol information available to PLECS without consuming any Flash space on the MCU.

Since this is not a SYS/BIOS application, the `F28xx_Headers_nonBIOS.cmd` file must be used in conjunction with `F28xx.cmd`.

Initialization and Task Dispatching

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

- `main.c/h` – main() routine, hardware and software interrupt routines.
- `io.c/h` – Initialization of inputs and outputs.

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.

Note the code also includes an inert `ControlTask2()` and `ControlBackground()`, primarily serving to illustrate multi-threading concepts and synchronization of multiple threads at the onset of a PIL simulation, according to "Task Synchronization at Start of Simulation" (on page 38).

The files related to the control algorithms are:

- `calib.c/h` – Control calibrations (settings).
- `pu.c/h` – Fixed-point reference values.
- `macros.h` – Macros for fixed-point calculations.
- `control.c/h` – Control tasks.
- `plx_control_fpu(32).lib` – Fixed-point control library.

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

Communication Interface

The demo project utilizes the serial communication interface (SCI) for exchanging information with PLECS.

- `sci.c/h` – SCI communication driver configured for `GPIO28/29`. Includes the communication callback function.

PIL Functionality

These files enable the demo application for PIL simulation with PLECS.

- `pil.h` – PIL framework API.
- `pil_codeid.c/h` – Defines PIL constants according to "Configuration Constants" (on page 40).

- `pil_comm.c/h` – Communication callback function. See “Communication Callbacks” (on page 28).
- `pil_ctrl.c/h` – Control callback for stepping the control tasks during a PIL simulation. See “Control Callback” (on page 33).
- `pil_symbols.c` – Definitions of override and read probe attributes. See “Probes” (on page 22).
- `pil_framework_fpu(32).lib` – PIL framework library, compiled for `fpu32` floating point support.

plexim

electrical engineering software

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