plecs

THE SIMULATION PLATFORM FOR
POWER ELECTRONIC SYSTEMS

PIL-BLDC Demo for STM32 F4 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-BLDC Demo for STM32 F4 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 brushless-DC 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)
- Keil μVision v5.12 – Download from keil.com.

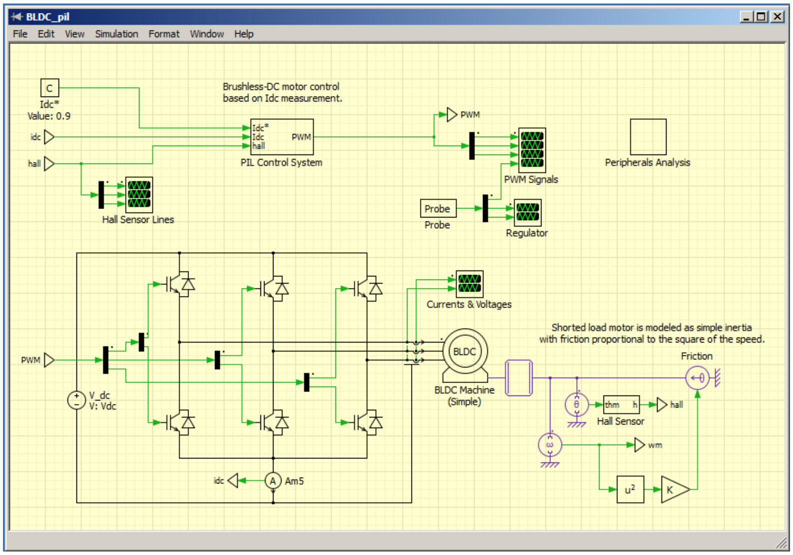
A license is required to run PLECS and activate the PIL package. You can request such a license from Plexim at plexim.com. Copy the license file `license.dat` that will be supplied to you into the directory in which you have installed PLECS.

1

Getting Started

This chapter provides a hands-on demonstration of how control-code executing on a STM32 F4xx 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 subsequent chapters.

The project is based on a basic current control application, with the embedded code controlling the switches of a three-phase inverter powering a brushless DC machine.



BLDC current control demo model

The sample code is designed to execute on a STM32 F4xx processor with either a serial VCP connected to USB or a native serial communication via US-

ART. The sample project actually supports the following boards and communication links:

- STM32F401 Nucleo using a STM32F401RET6 MCU (USART)
- STM32F411 Nucleo using a STM32F411RET6 MCU (USART)
- STM32F4 Discovery using a STM32F407VGT6 MCU (USART/VCP)
- STM32F4 Discovery using a STM32F429ZIT6U MCU (USART/VCP)

Further, the project can simply be adapted to work with other members of the STM32 F4 family. The pins/ports used for the communication link are shown in the table below.

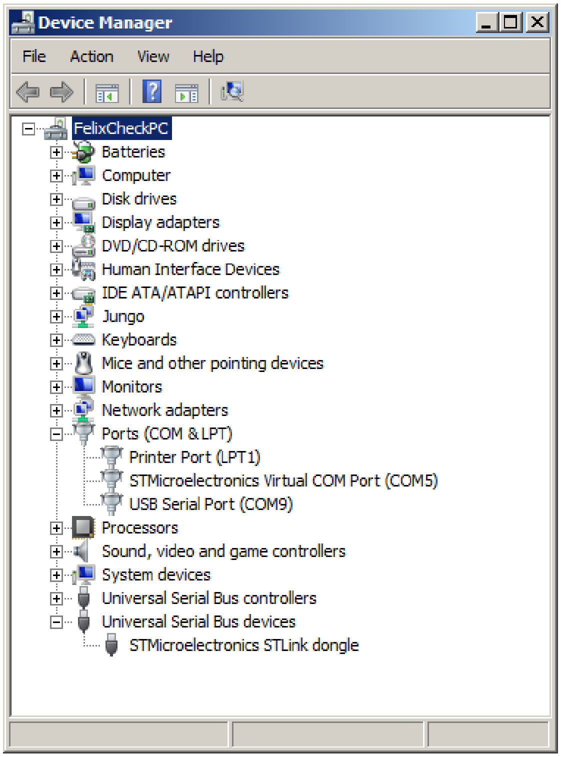
	VCP via USB	**USART TX/RX**
STM32 F401 Nucleo	x	GPIO PB6/PB7
STM32 F411 Nucleo	x	GPIO PB6/PB7
STM32 F407 Discovery	CN5	GPIO PB6/PB7
STM32 F407 Discovery	CN6	GPIO PB6/PB7

Ports used for the communication link

Note For a communication via USART, an additional USB to Serial Converter is required.

Programming the MCU

Connect the CN1 port of the MCU board to your PC. Also connect the communication cable depending on the desired communication link. Open the Windows Device Manager and confirm the enumeration of a COM port (either VCP or Serial) and a STLink Dongle.



- FelixCheckPC
  - Batteries
  - Computer
  - Disk drives
  - Display adapters
  - DVD/CD-ROM drives
  - Human Interface Devices
  - IDE ATA/ATAPI controllers
  - Jungo
  - Keyboards
  - Mice and other pointing devices
  - Monitors
  - Network adapters
  - Ports (COM & LPT)
    - Printer Port (LPT1)
    - STMicroelectronics Virtual COM Port (COM5)
    - USB Serial Port (COM9)
  - Processors
  - Sound, video and game controllers
  - System devices
  - Universal Serial Bus controllers
  - Universal Serial Bus devices
    - STMicroelectronics STLink dongle

COM port listed in device manager

The μVision Project containing the embedded code is provided within the PLECS distribution. Open the Project **STM32F4xx-BLDC_PIL_DEMO.uvprojx** and choose the desired MCU and communication link. Make sure to have the latest version of the PIL Tools installed (This can be seen in PLECS under File/PLECS Extensions/PIL) and build the project.

After a successful build, **Program** the MCU and press the **black Reset** button on the Board. If problems occur during programming, make sure that the **Debug Adapter Port** in the ST-Link Debugger Settings is chosen to **SW**.

Confirm that the green LED is blinking. The LED's on the different boards indicate the following system state:

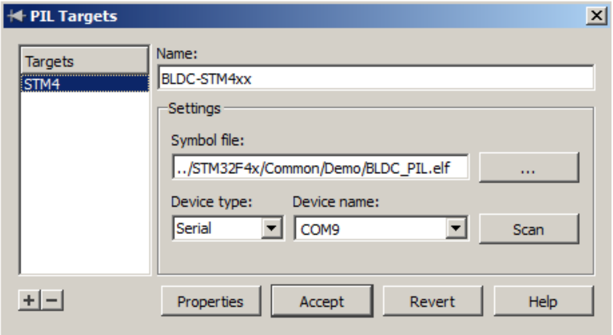
- Blinking Green LED: Algorithm executed (all boards)
- Orange LED: Communication Link ok (only F407 Discovery)
- Red LED: PWM Output stage Active (only F407 and F429 Discovery)
- Blue LED: PIL simulation running (only F407 Discovery)

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 **Windows** menu item **Target Manager**.



Targets: STM4

Name: BLDC-STM4xx

Symbol file: ../STM32F4x/Common/Demo/BLDC_PIL.elf

Device type: Serial

Device name: COM9

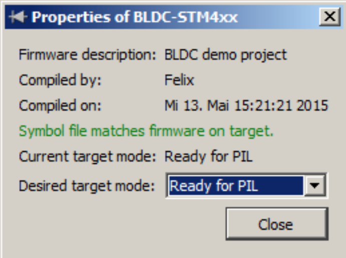
Target configuration

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 `BLDC_PIL.elf`.

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

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.



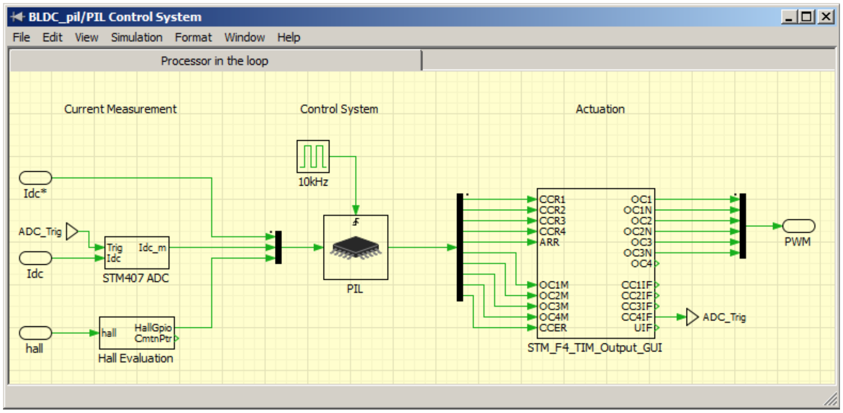
Firmware description: BLDC demo project
Compiled by: Felix
Compiled on: Mi 13. Mai 15:21:21 2015
Symbol file matches firmware on target.
Current target mode: Ready for PIL
Desired target mode: Ready for PIL

Target properties

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

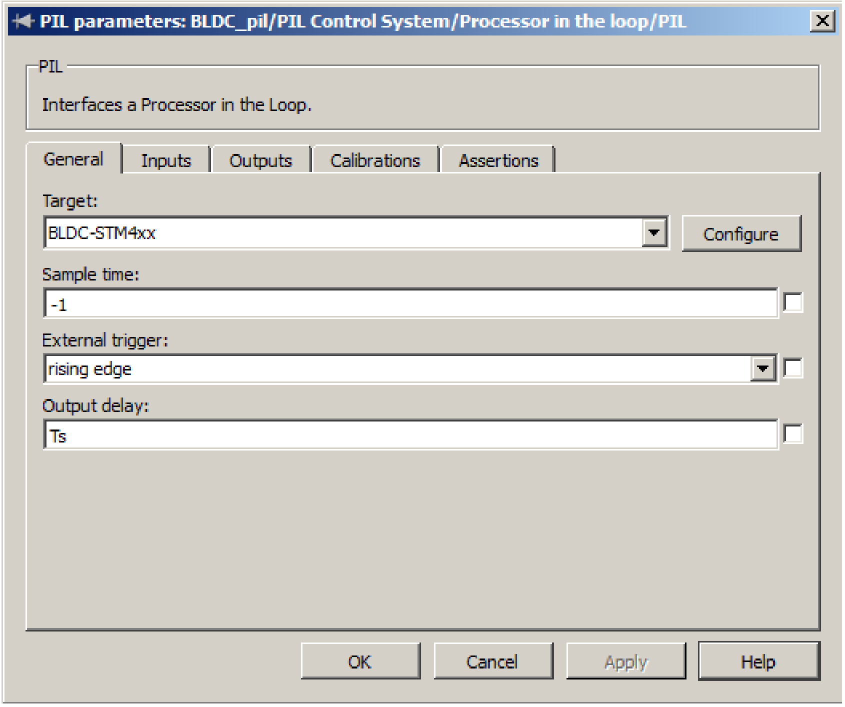
PIL Block

Now open the `BLDC_pil` model and look under the PIL Control System mask. 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 Pulse Generator.



BLDC control subsystem

Double-click on the PIL block. Select the target that you defined in the target manager from the **Target** combo box.



PIL parameters: BLDC_pil/PIL Control System/Processor in the loop/PIL

PIL

Interfaces a Processor in the Loop.

Target: BLDC-STM4xx

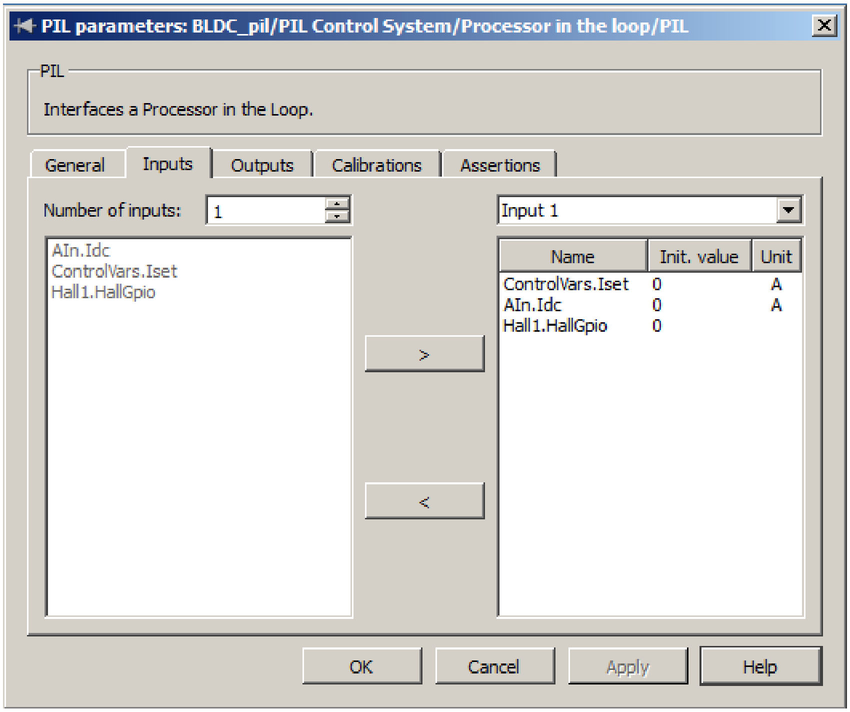
Sample time: -1 ☐

External trigger: rising edge ☐

Output delay: Ts ☐

PIL block general configuration

Activate the **Inputs** tab and see how the PIL block has been configured for one input.



PIL parameters: BLDC_pil/PIL Control System/Processor in the loop/PIL

PIL

Interfaces a Processor in the Loop.

Number of inputs: 1

AIn.Idc
ControlVars.Iset
Hall1.HallGpio

Input 1

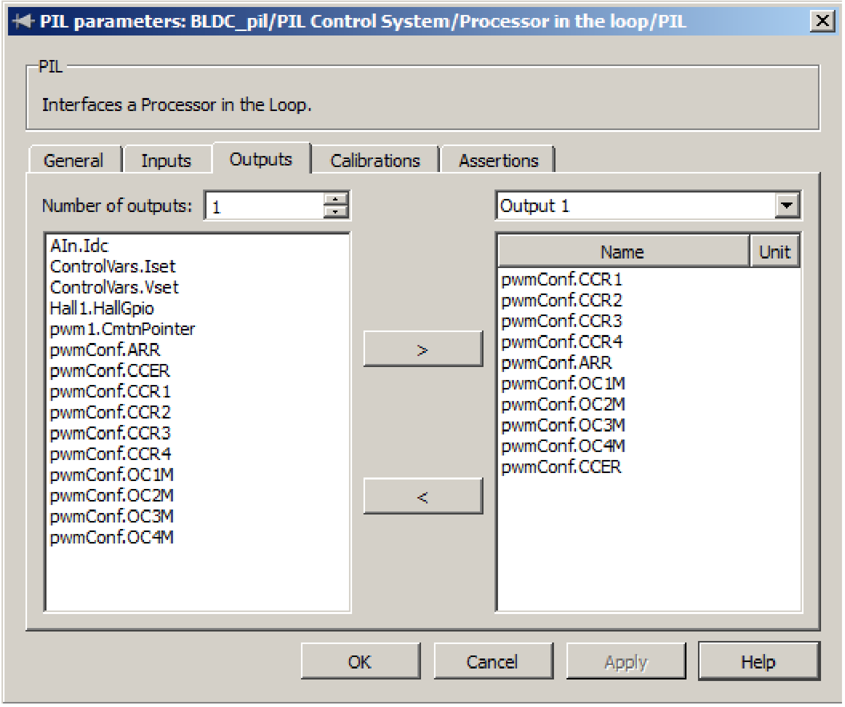
Name	Init. value	Unit
ControlVars.Iset	0	A
AIn.Idc	0	A
Hall1.HallGpio	0	

PIL block inputs

This input contains the following multiplexed signals:

- `ControlVars.Iset` – Direct current set-point (controlled by PI).
- `AIn.Idc` – DC Current measurement.
- `Hall1.Gpio` – Hall state for position measurement.

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 parameters: BLDC_pil/PIL Control System/Processor in the loop/PIL

PIL

Interfaces a Processor in the Loop.

Number of outputs: 1

Output 1

AIn.Idc
ControlVars.Iset
ControlVars.Vset
Hall1.HallGpio
pwm1.CmtnPointer
pwmConf.ARR
pwmConf.CCER
pwmConf.CCR1
pwmConf.CCR2
pwmConf.CCR3
pwmConf.CCR4
pwmConf.OC1M
pwmConf.OC2M
pwmConf.OC3M
pwmConf.OC4M

Name	Unit
pwmConf.CCR1	
pwmConf.CCR2	
pwmConf.CCR3	
pwmConf.CCR4	
pwmConf.ARR	
pwmConf.OC1M	
pwmConf.OC2M	
pwmConf.OC3M	
pwmConf.OC4M	
pwmConf.CCER	

PIL block outputs

The PIL block has been further configured for one output (**Outputs** tab) containing the following signals:

- `pwmConf.CCRx` – Timer peripheral compare register values.
- `pwmConf.ARR` – Timer period register.
- `pwmConf.OCxM` – Timer mode configuration registers.
- `pwmConf.CCER` – Timer output configuration register.

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 seven 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 the current flowing through the BLDC coils. Furthermore, see the current dips at the commutation events.

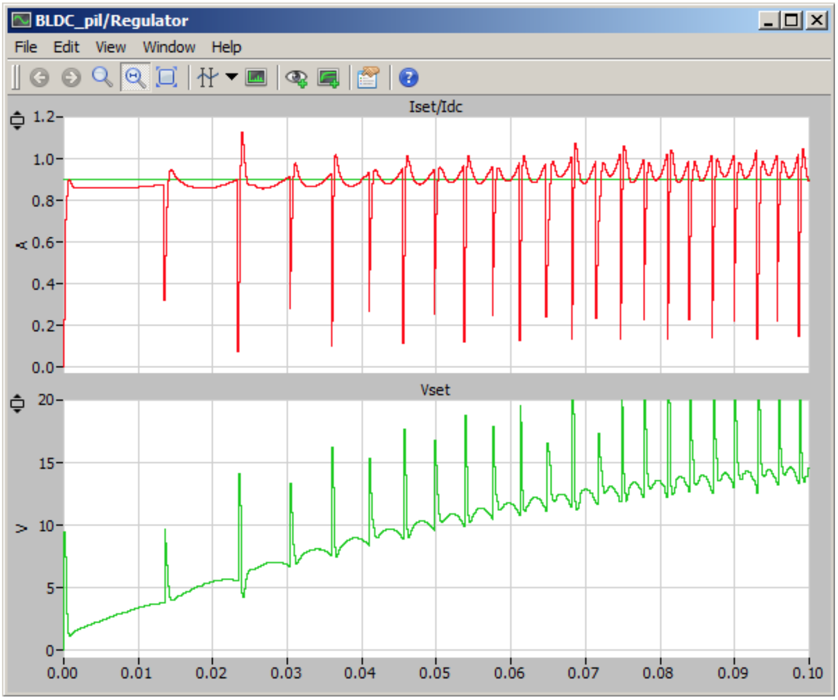


Figure 1.1: PIL Simulation Result

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

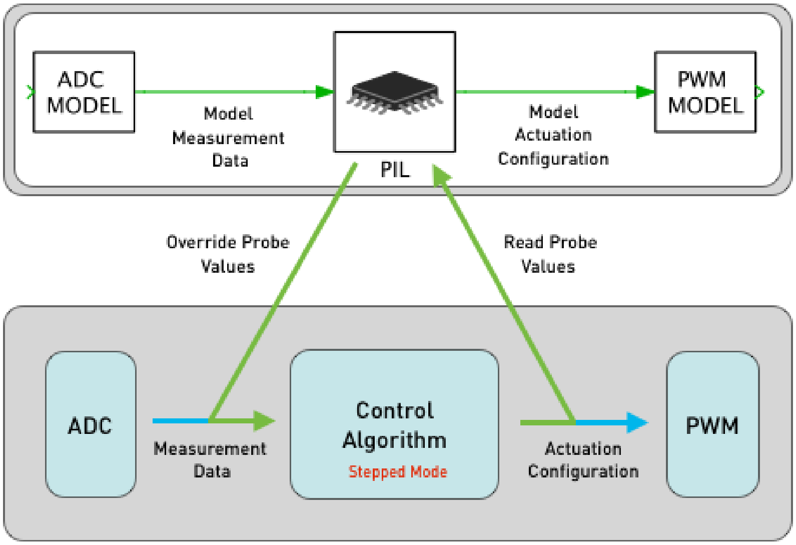
When developing embedded control algorithms, it is quite common to be testing such code, or portions thereof, by executing it inside a circuit simulator. Using PLECS, this can be easily achieved by means of a C-Script or DLL block. This approach is referred to as *Software-in-the-loop* (SIL). A SIL simulation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



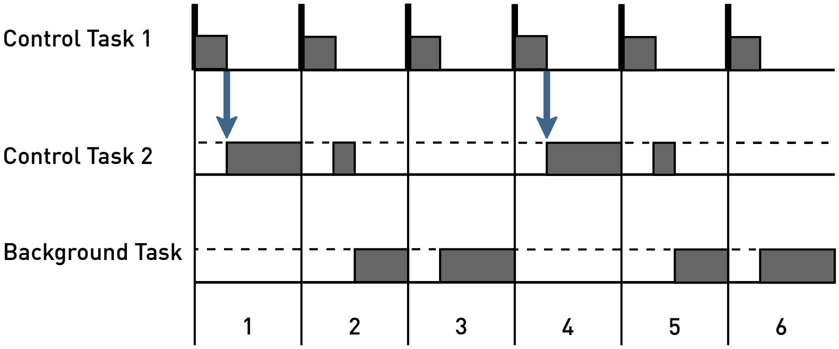
Principle of a PIL simulation

- 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*.

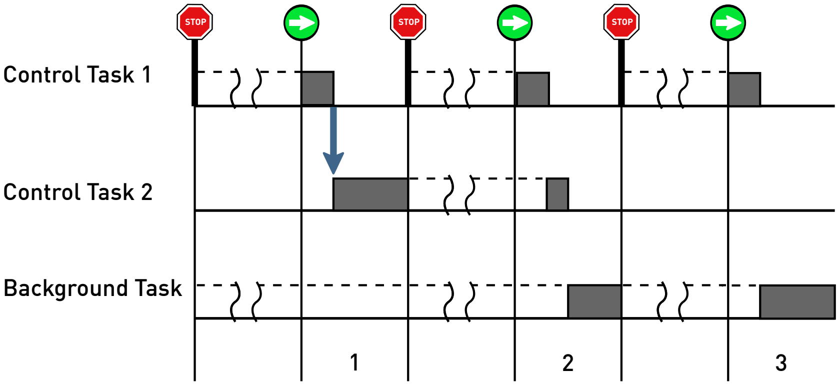
Let us further examine the pseudo real-time operation in the context of an embedded application utilizing nested control loops where fast high-priority tasks (such as current control) interrupt slower lower-priority tasks (such as voltage control). An example of such a configuration with two control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

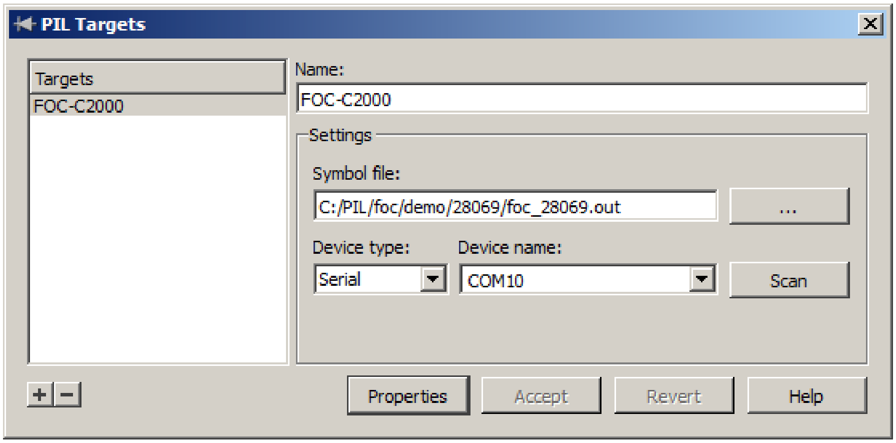
Once an embedded application is equipped with the PIL framework, and appropriate Override Probes and Read Probes are defined, it is ready for PIL simulations with PLECS.

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.



Targets: FOC-C2000

Name: FOC-C2000

Symbol file: C:/PIL/foc/demo/28069/foc_28069.out

Device type: Serial

Device name: COM10

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

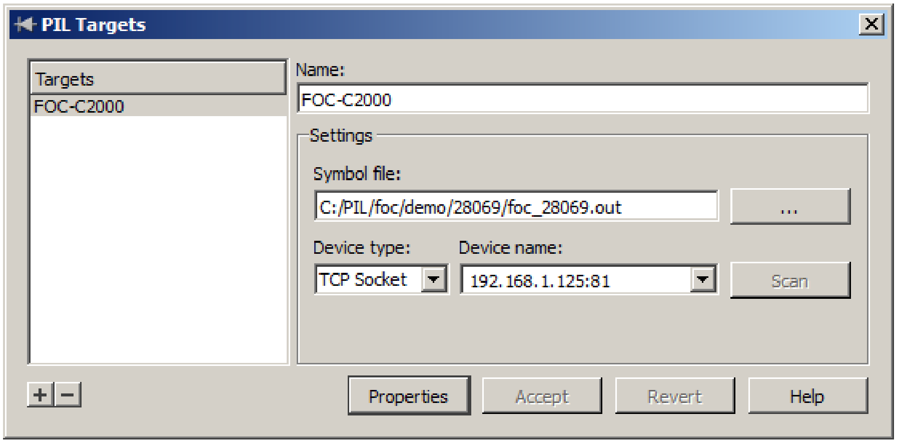
The **Serial device** selection corresponds to conventional physical or virtual serial communication ports. On a Windows machine, such ports are labeled `COMn`, where `n` is the number of the port.

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.



Name: FOC-C2000

Symbol file: C:/PIL/foc/demo/28069/foc_28069.out

Device type: TCP Socket

Device name: 192.168.1.125:81

TCP/IP Communication

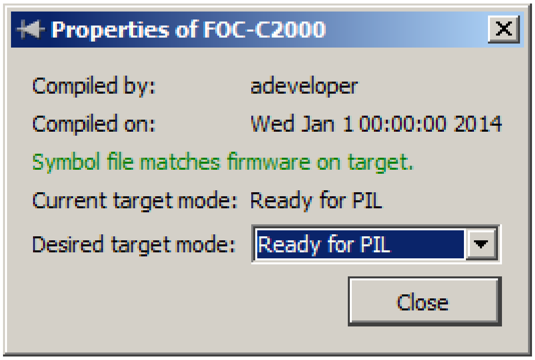
In this case the **Device name** corresponds to the IP address (or URL) and port number, separated by a colon (:).

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.


Properties of FOC-C2000

Compiled by: adeveloper

Compiled on: Wed Jan 1 00:00:00 2014

Symbol file matches firmware on target.

Current target mode: Ready for PIL

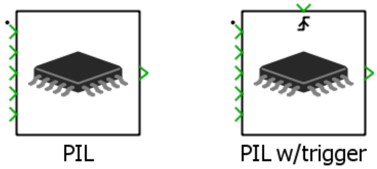
Desired target mode: Ready for PIL

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.

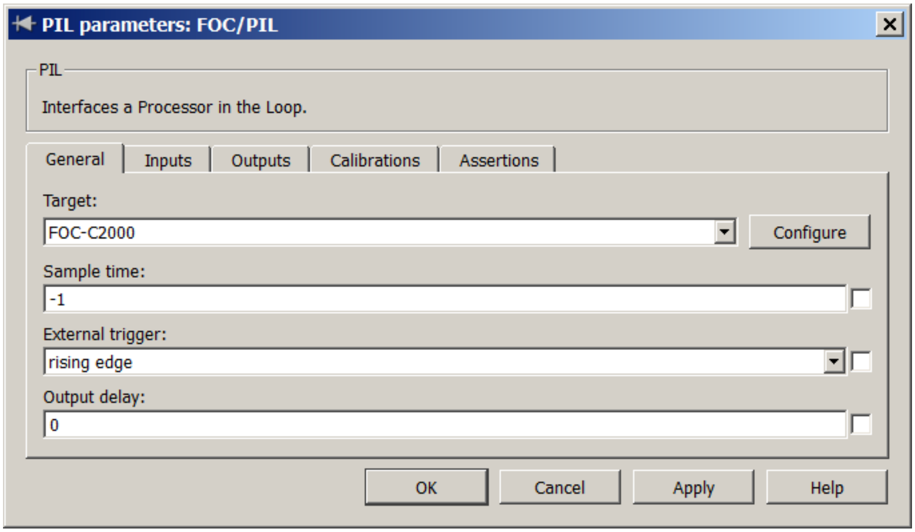
PIL Block

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 parameters: FOC/PIL

PIL

Interfaces a Processor in the Loop.

Target: FOC-C2000

Sample time: -1 ☐

External trigger: rising edge ☐

Output delay: 0 ☐

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]**.

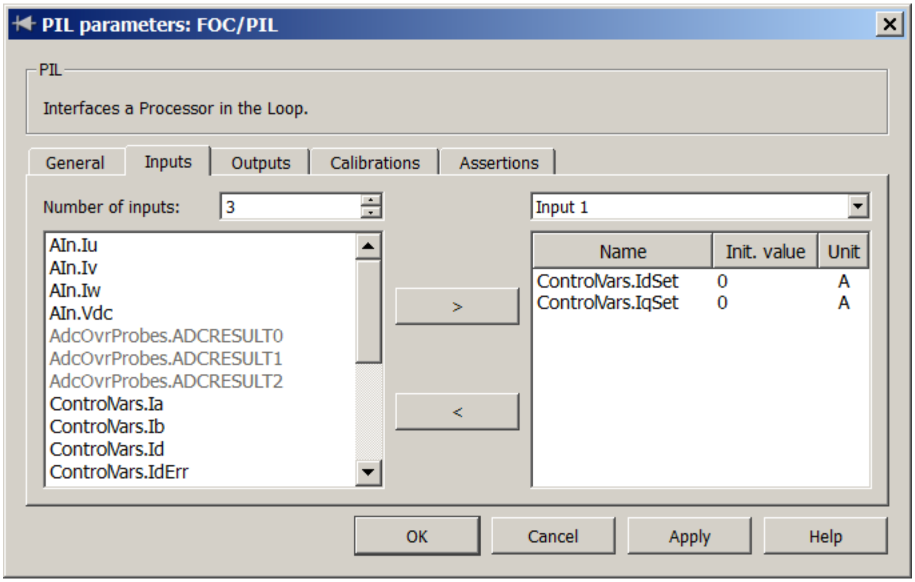
A trigger port can be enabled using the **External trigger** combo box. This is useful if the control interrupt source is part of the PLECS circuit, such as an ADC or PWM peripheral model.

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.



Interfaces a Processor in the Loop.

Number of inputs: 3

AIn.Iu
AIn.Iv
AIn.Iw
AIn.Vdc
AdcOvrProbes.ADCRESULT0
AdcOvrProbes.ADCRESULT1
AdcOvrProbes.ADCRESULT2
ControlVars.Ia
ControlVars.Ib
ControlVars.Id
ControlVars.IdErr

Input 1

Name	Init. value	Unit
ControlVars.IdSet	0	A
ControlVars.IqSet	0	A

PIL Block Inputs Tab

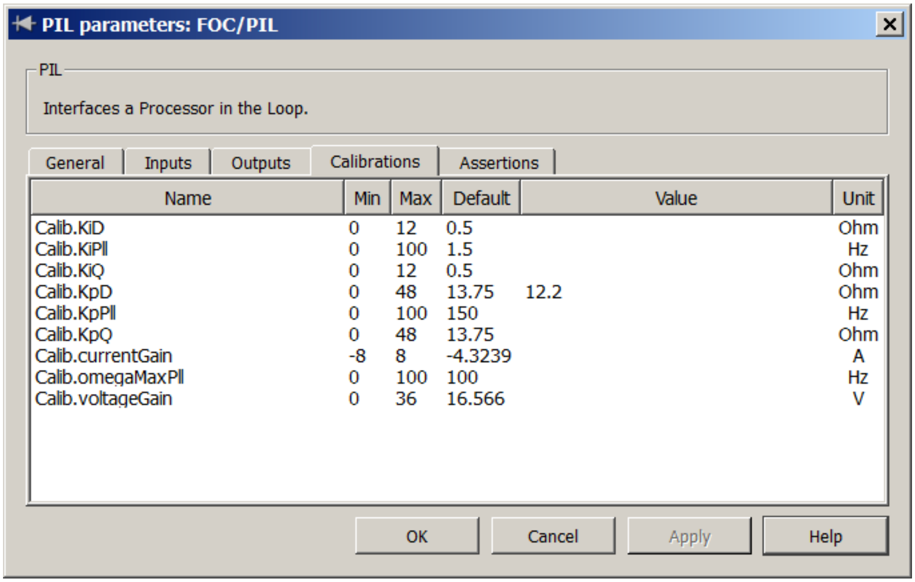
The PIL block extracts the names of Override Probes and Read Probes from the symbol file selected in the target configuration and presents lists for selection as input and output signals, as shown in the figure above.

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



Interfaces a Processor in the Loop.

Name	Min	Max	Default	Value	Unit
Calib.KiD	0	12	0.5		Ohm
Calib.KiPll	0	100	1.5		Hz
Calib.KiQ	0	12	0.5		Ohm
Calib.KpD	0	48	13.75	12.2	Ohm
Calib.KpPll	0	100	150		Hz
Calib.KpQ	0	48	13.75		Ohm
Calib.currentGain	-8	8	-4.3239		A
Calib.omegaMaxPll	0	100	100		Hz
Calib.voltageGain	0	36	16.566		V

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.

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.

```
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:

```
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 $\frac{512}{2^{10}} * 5\mathrm{V} = 2.5\mathrm{V}$.

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.

```
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 $0.5 * 1800\mathrm{rpm} = 900\mathrm{rpm}$.

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

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

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.

```
struct BATTERY {
  PIL_OVERRIDE_PROBE(int16_t, voltage, 10, 5.0, "V");
};

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.

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

```
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.

```
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 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.

```
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.

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.

```
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.

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

```
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.

```
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

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.

```
/**
 * 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

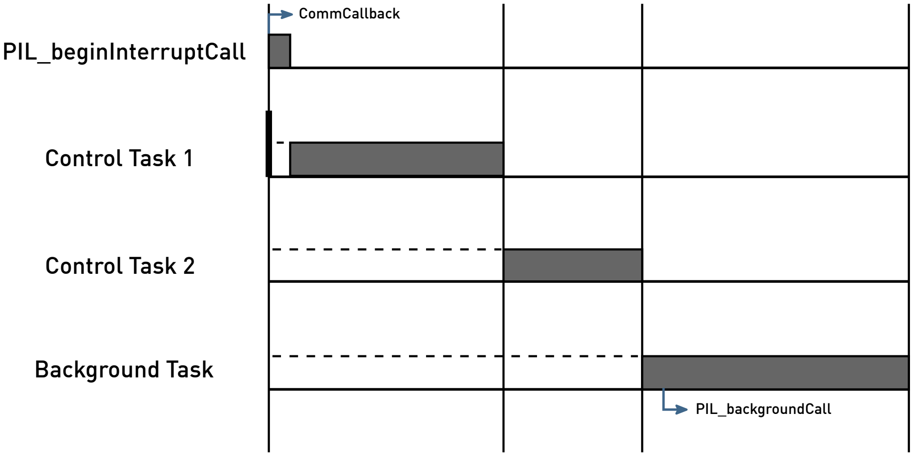
	Real-time	**Pseudo Real-time**
PIL_beginInterruptCall	CommCallback	CommCallback BackgroundCommClbk Message Evaluation PIL Cmd Handling
PIL_backgroundCall	BackgroundCommClbk Message Evaluation PIL Cmd Handling	N/A

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.

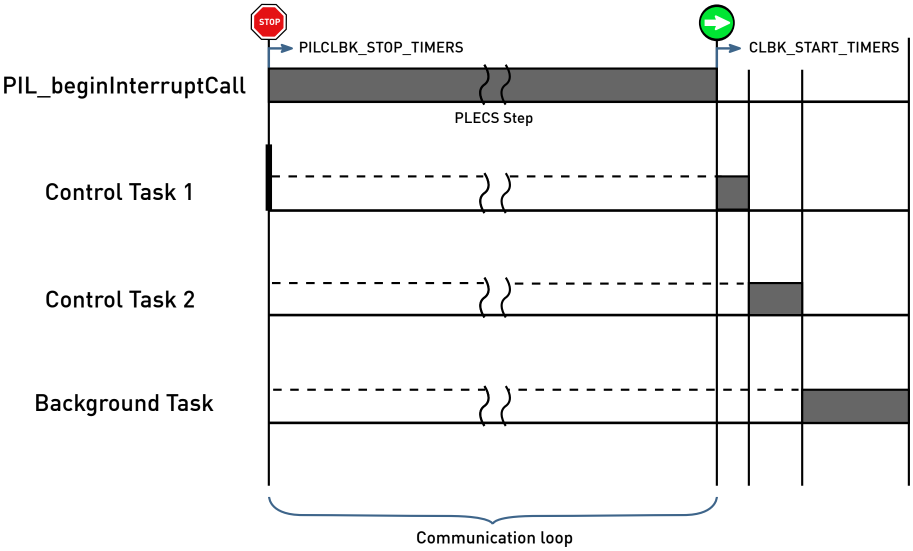


PIL framework during real-time operation

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.

```
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;
   }
}
```

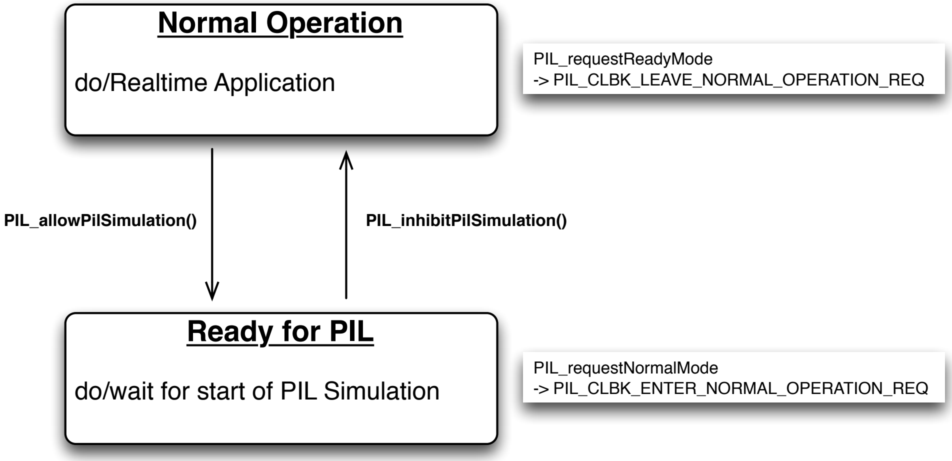
- `PIL_CLBK_INITIALIZE_SIMULATION` – Called at the beginning of a PIL simulation. Used to reset the controller(s) and control task dispatching to initial conditions.
- `PIL_CLBK_TERMINATE_SIMULATION` – Called at the end of a PIL simulation.
- `PIL_CLBK_STOP_TIMERS` – Called at the beginning of the control 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


Normal Operation
do/Realtime Application
PIL_requestReadyMode
-> PIL_CLBK_LEAVE_NORMAL_OPERATION_REQ
PIL_allowPilSimulation()
PIL_inhibitPilSimulation()
Ready for PIL
do/wait for start of PIL Simulation
PIL_requestNormalMode
-> PIL_CLBK_ENTER_NORMAL_OPERATION_REQ

PIL target modes and mode change requests

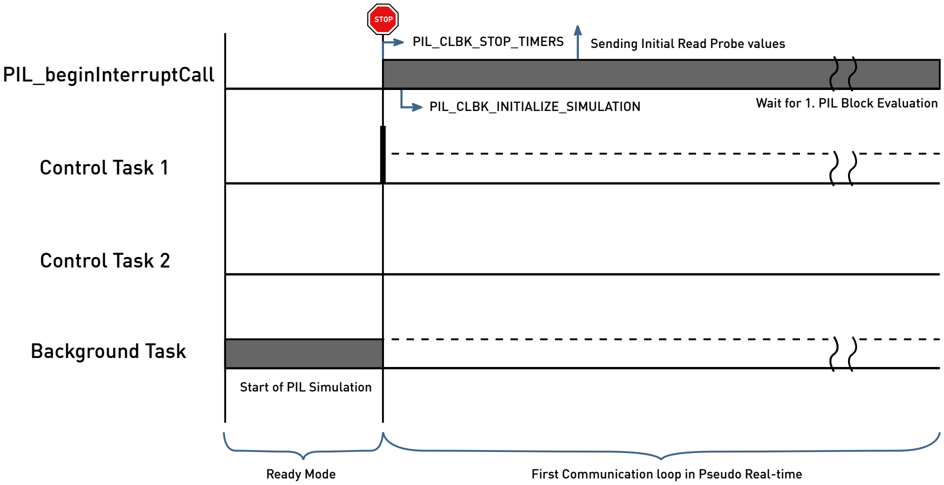
in this mode due to the power stage being active. A PIL simulation can only be started if the target is in Ready for PIL mode, which corresponds to a safe state in which the power stage is disabled. As explained in the prior section, the code for enabling or disabling the power stage is application specific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


STOP
PIL_CLBK_STOP_TIMERS
Sending Initial Read Probe values
PIL_beginInterruptCall
PIL_CLBK_INITIALIZE_SIMULATION
Wait for 1. PIL Block Evaluation
Control Task 1
Control Task 2
Background Task
Start of PIL Simulation
Ready Mode
First Communication loop in Pseudo Real-time

Start of a PIL Simulation

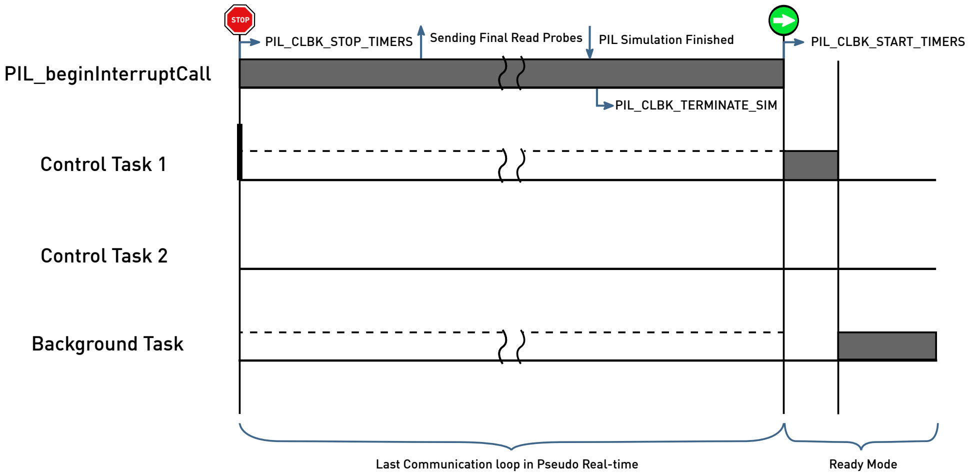
Note The initial conditions of Read Probes are fed into the PLECS model at simulation time t=0. However, these values will be immediately modified if the PIL block is also triggered at time t=0 and the output delay of the block is set to zero.

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

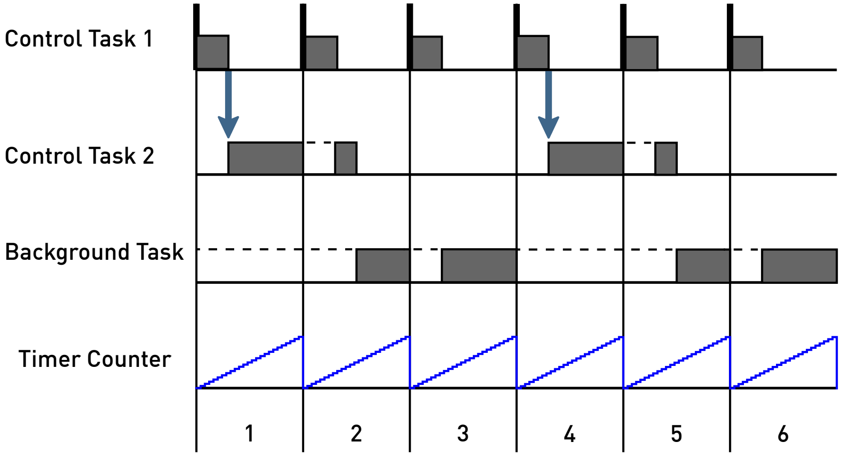
Control Dispatching

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.



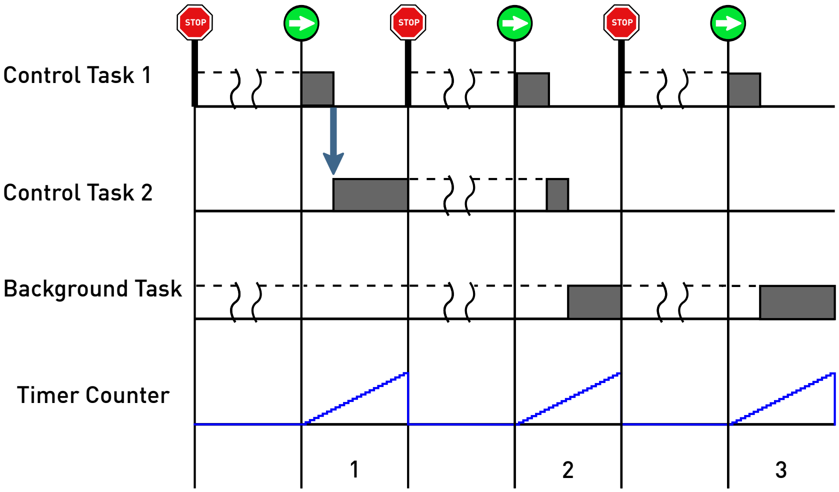
End of a PIL Simulation



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.

```
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

```
  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.

```
PIL_CONST_DEF(unsigned char, Guid[], CODE_GUID);
PIL_CONST_DEF(unsigned char, CompiledDate[], COMPILE_TIME_DATE_STR);
PIL_CONST_DEF(unsigned char, CompiledBy[], USER_NAME);

PIL_CONST_DEF(uint32_t, BaudRate, BAUD_RATE);
PIL_CONST_DEF(uint16_t, StationAddress, 0);
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.

```
PIL_CONFIG_DEF(uint32_t, SysClk, SYSCLK_HZ);
PIL_CONFIG_DEF(uint32_t, PwmFrequency, PWM_HZ);
PIL_CONFIG_DEF(uint32_t, ControlFrequency, CONTROL_HZ);
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;
```

STM32 F4xx 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.

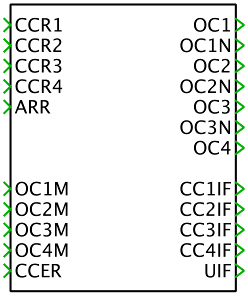
The PLECS PIL library includes high-fidelity MCU peripheral models which work at the register level, and are therefore well-suited for PIL simulations. Furthermore, certain blocks have a second implementation with a graphical user interface (GUI) that automatically determines the register configurations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

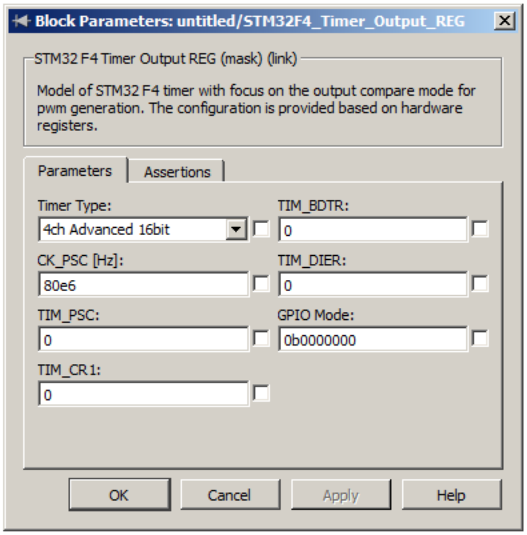
Subsequent sections describe the PLECS peripheral components in detail and highlight modeling assumptions and limitations. When documenting peripheral register settings, the following color coding is used:

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

System Timer for PWM Generation (Output Mode)

The PLECS peripheral library provides two blocks for the STM32 F4 system timer used in output mode. One block has a register-based configuration mask and a second block features a GUI. In both cases, you should distinguish between registers configured in the parameter mask and inputs to the block. Mask parameters are fixed (static) during a simulation and correspond to the configurations which the embedded software uses 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 GUI, 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.




Block Parameters: untitled/STM32F4_Timer_Output_REG

STM32 F4 Timer Output REG (mask) (link)

Model of STM32 F4 timer with focus on the output compare mode for pwm generation. The configuration is provided based on hardware registers.

Timer Type: 4ch Advanced 16bit ☐

CK_PSC [Hz]: 80e6 ☐

TIM_PSC: 0 ☐

TIM_CR1: 0 ☐

TIM_BDTR: 0 ☐

TIM_DIER: 0 ☐

GPIO Mode: 0b0000000 ☐

Register-based Timer model for output mode

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.

Timer Subtypes

The STM32 F4 MCU's provide several subtypes of timers which can be used for input capture, output compare and PWM generation functionalities. In the presented model, all subtypes listed below are combined in one module and can be chosen via the component mask:

- 4 Channel 16bit Advanced Timer
- 4 Channel 32bit General Purpose Timer
- 4 Channel 16bit General Purpose Timer
- 2 Channel 16bit General Purpose Timer
- 1 Channel 16bit General Purpose Timer

The focus of this model is the timer output behavior meaning that all input functionalities are disregarded. This corresponds to the hardware behavior with all TIM_CCMRx.CCyS cells being set to 00. Further, the One-Shot mode of the module is not supported. In the following sections, the common part of all subtypes is explained together with the models limitations. Further, the differences between the subtypes are described in more detail.

General Counter Behavior

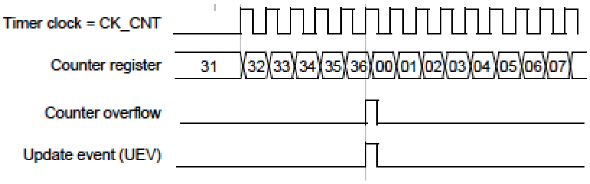
The base of all timer modules is an auto-reload counter driven by a prescaled counter clock *CK_CNT*. The period of this time base clock is determined by the counter clock frequency *CK_PSC* and the prescaler register *TIM_PSC*, both configurable in the mask, as follows:

$$T_{CK\_CNT} = \frac{TIM\_PSC + 1}{CK\_PSC}$$

The counter either operates in Edge-aligned mode with configurable direction or in Center-aligned mode. In addition to the general counter functionality, the module also generates output compare interrupt flags when the counter matches the values stored in the *CCRx* registers. Those flags are later used to determine the output levels of the timer module.

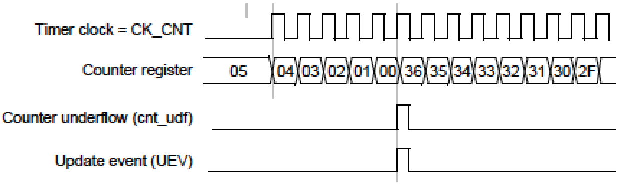
Edge-aligned mode

In upcounting direction, the counter counts from 0 to the counter period value *TIM_ARR* and generates an update event *UEV* simultaneous to the counter overflow.



Edge-aligned mode / Upcounting [1]

In downcounting direction, the counter counts from *TIM_ARR* to 0 and generates an update event (*UEV*) simultaneous to the counter underflow.



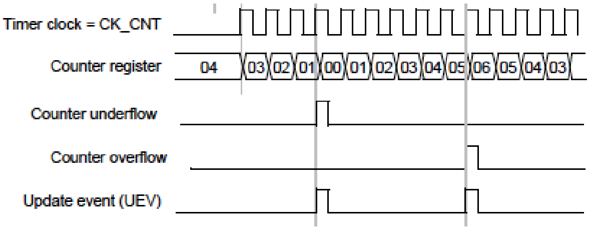
Edge-aligned mode / Downcounting [1]

In Edge-aligned mode, the counter period and therefore the PWM period is calculated as:

$$T_{PWM} = T_{CK\_CNT} \cdot (TIM\_ARR + 1)$$

Center-aligned mode

In this mode, the counter alternates its direction and generates an update event (*UEV*) at the counter under- and overflow. In the model, the counter always starts in upcounting direction.

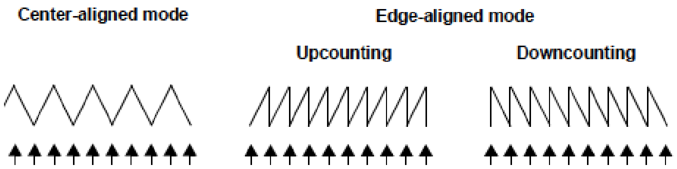


Center-aligned mode [1]

For Center-aligned mode, the PWM period is calculated as:

$$T_{PWM} = T_{CK\_CNT} \cdot 2 \cdot TIM\_ARR$$

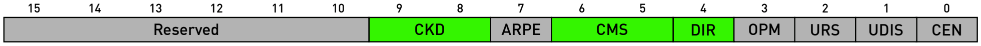
For all modes, the timer model operates in preloaded mode, meaning that the used configuration is updated simultaneously to the update events. The Repetition Counter functionality is not supported in the model.



Events used for configuration update [1]

In other words, all input terminals of the model, except the *CCER* register, are sampled with the instants of the update events.

The timer mode, direction and output compare flag behavior can be set jointly using the *TIM_CR1* register.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD		ARPE	CMS		DIR	OPM	URS	UDIS	CEN

Timer Mode Configuration

The *CKD* field only has an effect on the subtypes with PWM dead time generation and is therefore described in a later section. The register cell *CMS* can be used to determine the counter mode and the output compare flag behavior.

- 00 - Edge-aligned mode
- 01 - Center-aligned mode 1 - compare flags only set when counting down
- 10 - Center-aligned mode 2 - compare flags only set when counting up
- 11 - Center-aligned mode 3 - compare flags set when counting up and down

In Edge-aligned mode, the *DIR* bit determines the counter direction.

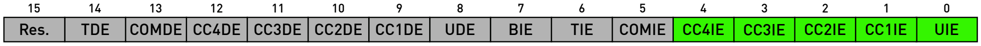
- 0 - Upcounting
- 1 - Downcounting

The module assumes the timer as always active and to be operated in preloaded mode with the update event generation always enabled. Therefore, the following settings are mandatory when using the register-based version.

- TIM_CR1.ARPE = 1
- TIM_CR1.UDIS = 0
- TIM_CR1.CEN = 1

Interrupt Flags

The timer module can generate interrupt flags at the *CCxIF* and *UIF* output terminals. Those flags are based on the counter compare and update event flags and can be used in the model to, i.e., trigger an ADC conversion or a new control step via the PIL block. Note that in the model those flags are implemented as pulses.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TDE	COMDE	CC4DE	CC3DE	CC2DE	CC1DE	UDE	BIE	TIE	COMIE	CC4IE	CC3IE	CC2IE	CC1IE	UIE

Interrupt enable register

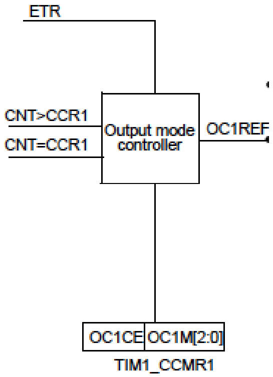
The interrupt flags can be enabled with the bits of the *TIM_DIER* register.

- 0 - interrupt disabled
- 1 - interrupt enabled

Note Only the four channel subtype implementations make use of all *CCxIE* fields.

Output Mode Controller

The output-mode controller generates up to 4 reference signals *OCyREF* based on the output compare flags of the counter.



Output Mode Controller for *OCyREF* [1]

The controller implements several output modes defining the behavior of *OCyREF*. With the register fields *TIM_CCMRx.OCyM*, the mode of each reference signal can be specified separately.

- 000 - Frozen, comparisons have no effect on *OCyREF*
- 001 - Active match mode, *OCyREF* forced high when $CTR = CCRy$
- 010 - Inactive match mode, *OCyREF* forced low when $CTR = CCRy$
- 011 - Toggle mode, *OCyREF* toggled when $CTR = CCRy$
- 100 - Force inactive mode, *OCyREF* always forced low
- 101 - Force active mode, *OCyREF* always forced high
- 110 - PWM Mode 1
- 111 - PWM Mode 2

Because the reference signal mode is supposed to be changed during simulation, the OCyM fields can be accessed via the input terminals. Note that those are also updated with the update events generated by the timer.

The hardware options to externally clear the reference signal are not supported in the model. Further, the break function of the timer is not part of the model assuming the flag *BDTR.MOE* is always set. Therefore it is mandatory to set *MOE* to 1 while using the resister-based version.

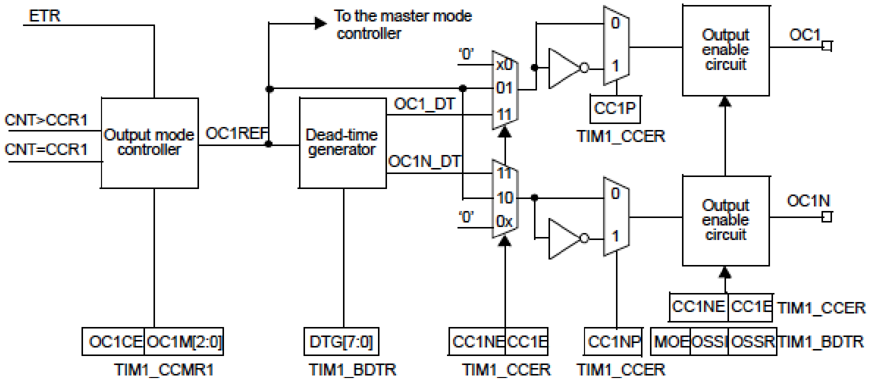
The options available in the output stage majorly depend on the timer subtype and therefore are discussed in the subsequent sections. The configuration of all output stages is done with the *CCER* register.

Note The *CCER* is accessed via the input terminals and is not preloaded. This means that a change on the *CCER* input directly effects the outputs.

---

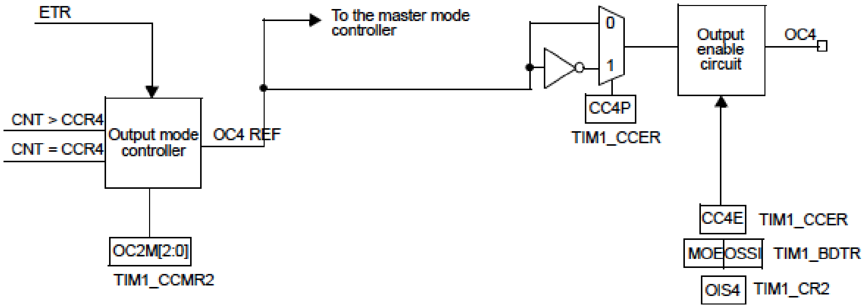
4 channel Advanced Timer

The Advanced Timer consists of a timer and a 4 channel output stage. The timer has a width of 16-bit and can be operated in Edge-aligned (up and down) as well as Center-aligned mode. For channels 1 to 3, the output stage enables complementary outputs with dead time and configurable polarity.



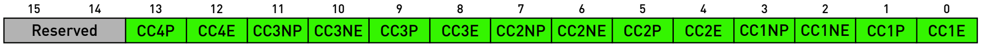
Output stage of Advanced Timer (channel 1 to 3) [1]

For channel 4, the output stage shown below only supports configurable polarity.



Output stage of Advanced Timer (channel 4) [1]

The *CCER* register can be used to configure all channels of the output stage separately.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		CC4P	CC4E	CC3NP	CC3NE	CC3P	CC3E	CC2NP	CC2NE	CC2P	CC2E	CC1NP	CC1NE	CC1P	CC1E

Channel-wise configuration of output stage

With the CCxP and CCxNP fields, the polarity of the output signal can be inverted.

- 0 - Active High (regular polarity)
- 1 - Active Low (inverted polarity)

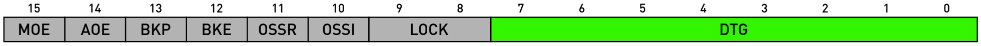
With the CCxE and CCxNE bits, the output can be enabled.

- 0 - Output Enabled, OCx/OCxN defined by OCxRef
- 1 - Output Disabled, OCx/OCxN defined by GPIO Mode

Those bits further effect the output stage behavior for channels 1 to 3. The table below shows this for both outputs operated with equal polarity.

CCxNE	CCxE	Behavior
0	0	OCx & OCxN inactive
0	1	OCx = OCxRef, OCxN inactive
1	0	OCx inactive, OCxN = OCxRef
1	1	Complementary output mode with dead time

The dead time for each positive flank in *OCx* and *OCNx* is configured with the *TIM_BDTR* register.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOE	AOE	BKP	BKE	OSSR	OSSI	LOCK		DTG							

Dead time configuration

The dead time (DT) can be calculated based on the cell *DTG* as shown below. The bits *DTG[7:5]* determine the formula used for its calculation.

- $0xx$ - $DT = DTG[7:0] \cdot t_{dtg}$ with $t_{dtg} = t_{DTS}$
- $10x$ - $DT = (64 + DTG[5:0]) \cdot t_{dtg}$ with $t_{dtg} = 2 \cdot t_{DTS}$
- 110 - $DT = (32 + DTG[4:0]) \cdot t_{dtg}$ with $t_{dtg} = 8 \cdot t_{DTS}$
- 111 - $DT = (32 + DTG[4:0]) \cdot t_{dtg}$ with $t_{dtg} = 16 \cdot t_{DTS}$

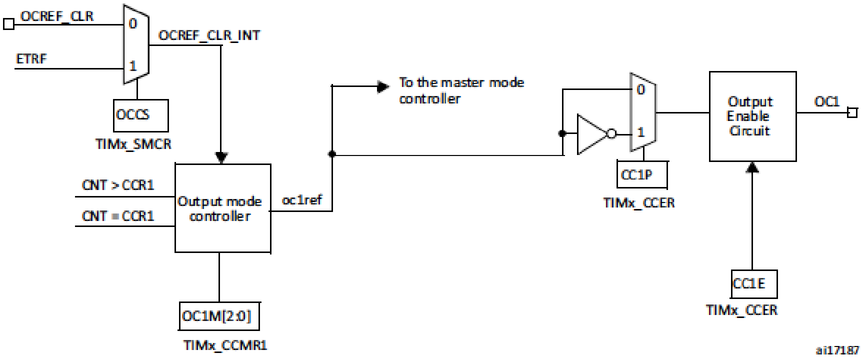
The dead time clock t_{DTS} is related to the timer clock period T_{CK_CNT} and can be configured with the field *CKD* of the *TIM_CR1* register.

- 00 - $t_{DTS} = T_{CK\_CNT}$
- 01 - $t_{DTS} = 2 \cdot T_{CK\_CNT}$
- 10 - $t_{DTS} = 4 \cdot T_{CK\_CNT}$
- 11 - not supported

This subtype implementation uses the full set of inputs, outputs and configuration registers.

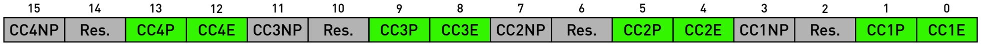
4 channel General Purpose Timer

This subtype is available with a 16-bit or 32-bit counter implementation both supporting Edge-aligned (up and down), as well as Center-aligned modes. The 4 channel output stage shown below only supports configurable polarity.



Output stage of general purpose timer (channel 1/4) [1]

The *CCER* register can be used to configure all channels of the output stage separately.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CC4NP	Res.	CC4P	CC4E	CC3NP	Res.	CC3P	CC3E	CC2NP	Res.	CC2P	CC2E	CC1NP	Res.	CC1P	CC1E

Channel-wise configuration of output stage

With the CCxP bits, the polarity of the output signal can be inverted.

- 0 - Active High (regular polarity)

- 1 - Active Low (inverted polarity)

With the CCxE and CCxNE bits, the output can be enabled.

- 0 - Output Enabled, OCx/OCxN defined by OCxRef
- 1 - Output Disabled, OCx/OCxN defined by GPIO Mode

Note The CCxNP bits have no effect on the model.

The terminals used by this subtype are shown in the table below.

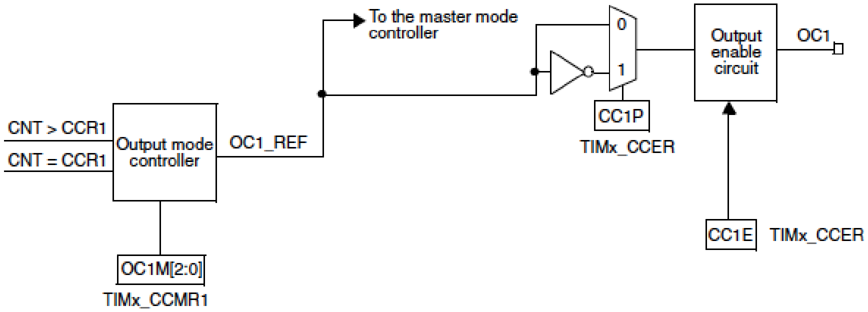
Terminal Group	Utilized	Unused
Input	CCR1 - CCR4, ARR, CCER, OC1M - OC4M	x
Output	OC1 - OC4, CC1IF-CC4IF, UEV	OC1N - OC3N

Unused mask registers, register cells and further limitations are listed below.

- TIM_BDTR
- TIM_CR1.CKD
- GPIO Mode for unused outputs

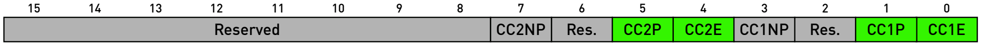
2 channel General Purpose Timer

This subtype contains a 16-bit counter only supporting Edge-aligned, Upcounting mode. The 2 channel output stage shown below only supports configurable polarity.



Output stage of general purpose timer (channel 1/2) [1]

The *CCER* register can be used to configure both channels of the output stage separately.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								CC2NP	Res.	CC2P	CC2E	CC1NP	Res.	CC1P	CC1E

Channel-wise configuration of output stage

With the CCxP bits, the polarity of the output signal can be inverted.

- 0 - Active High (regular polarity)
- 1 - Active Low (inverted polarity)

With the CCxE and CCxNE bits, the output can be enabled.

- 0 - Output Enabled, OCx/OCxN defined by OCxRef
- 1 - Output Disabled, OCx/OCxN defined by GPIO Mode

Note The CCxNP bits have no effect on the model.

The terminals used by this subtype are shown in the table below.

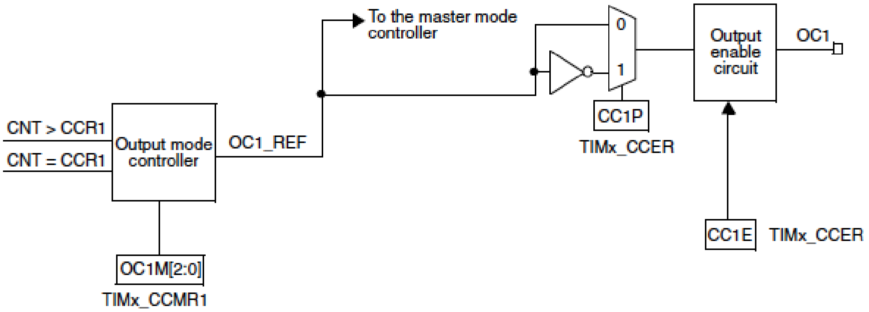
Terminal Group	Utilized	Unused
Input	CCR1 - CCR2, ARR, CCER, OC1M - OC2M	CCR3 - CCR4, OC3M-OC4M
Output	OC1 - OC2, CC1IF - CC2IF, UEV	OC3 - OC4, OC1N - OC3N, CC3IF - CC4IF

Unused mask registers, register cells and further limitations are listed below.

- TIM_BDTR
- TIM_CR1.CKD
- TIM_DIER.CC3IE - TIM_DIER.CC4IE
- GPIO Mode for unused outputs
- TIM_CR1.CMS only supports 00
- TIM_CR1.DIR only supports 0

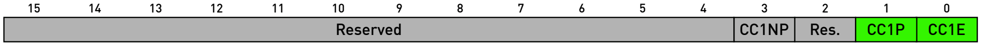
1 channel General Purpose Timer

This subtype contains a 16-bit counter only supporting Edge-aligned, Upcounting mode. The single channel output stage shown below only supports configurable polarity.



Output stage of general purpose timer (channel 1/1) [1]

The *CCER* register can be used to configure single channel output stage.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												CC1NP	Res.	CC1P	CC1E

Configuration of the output stage

With the CCxP bits, the polarity of the output signal can be inverted.

- 0 - Active High (regular polarity)
- 1 - Active Low (inverted polarity)

With the CCxE and CCxNE bits, the output can be enabled.

- 0 - Output Enabled, OCx/OCxN defined by OCxRef
- 1 - Output Disabled, OCx/OCxN defined by GPIO Mode

Note The CC1NP bit has no effect on the model.

The terminals used by this subtype are shown in the table below.

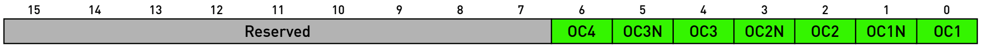
Terminal Group	Utilized	Unused
Input	CCR1, ARR, CCER, OC1M	CCR2 - CCR4, OC2M-OC4M
Output	OC1, CC1IF, UEV	OC2 - OC4, OC1N - OC3N, CC2IF - CC4IF

Unused mask registers, register cells and further limitations are listed below.

- TIM_BDTR
- TIM_CR1.CKD
- TIM_DIER.CC2IE - TIM_DIER.CC4IE
- GPIO Mode for unused outputs
- TIM_CR1.CMS only supports 00
- TIM_CR1.DIR only supports 0

GPIO Mode

In case that an output enable circuit is configured as inactive, the output level is determined by the GPIO Mode. To mimic this in the simulation model, the parameter GPIO Mode is available in the register-based version.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved									OC4	OC3N	OC3	OC2N	OC2	OC1N	OC1

Configuration of GPIO Mode

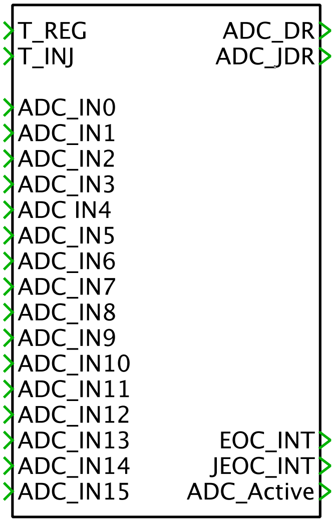
With the bits *OCx* and *OCxN*, the corresponding output mode can be set.

- 0 - Pull-Down (Inactive Low)
- 1 - Pull-Up (Inactive High)

Note This Register is available only in the simulation.

Analog-Digital Converter (ADC)

The PLECS peripheral library provides two blocks for the STM32 F4 single ADC module, one with a register-based configuration mask and a second with a GUI. The figure below shows the appearance of the block.



ADC module model

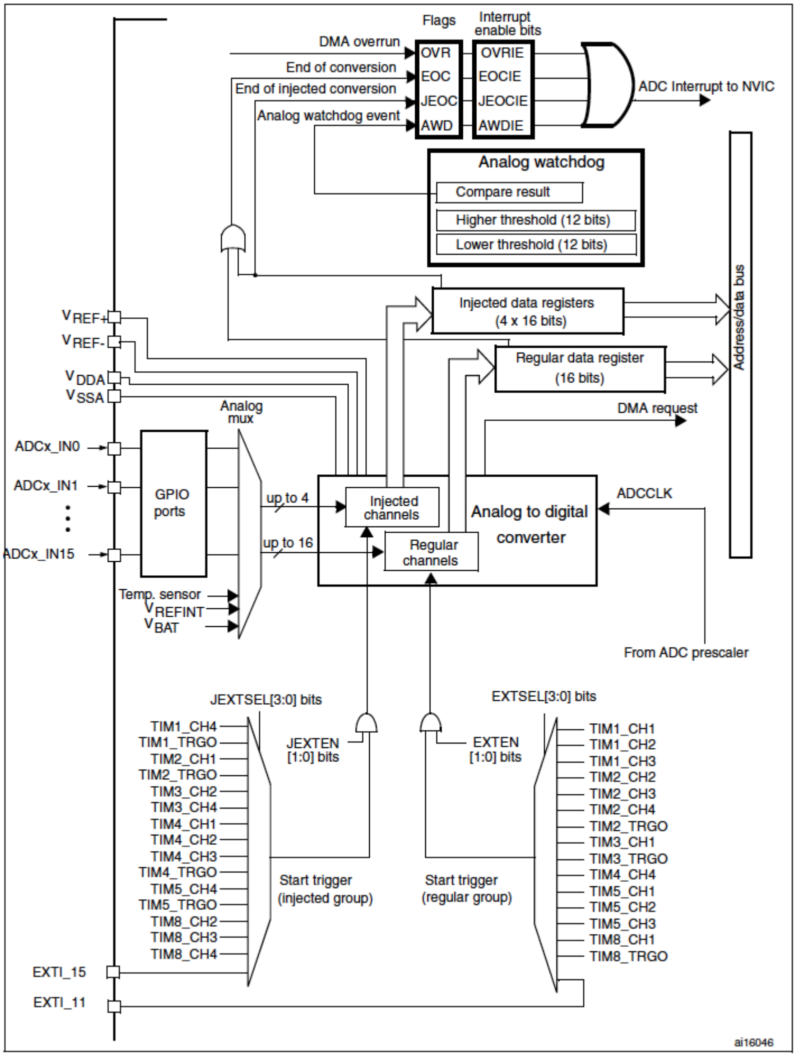
The register-based version allows the user to directly enter register values in decimal, binary, or hexadecimal notation. For convenience, the peripheral library also provides a component with a GUI to simplify the configuration.

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

- *T_REG,T_INJ* - input ports to trigger ADC conversions
- *ADC_INx* - input measurement channels
- *ADC_DR* - auto-size output port to access regular conversion results
- *ADC_JDR* - auto-size output port to access injected conversion results
- *xEOC_INT* - output ports for subsequent logic triggered by a conversion end
- *ADC_Active* - output port indicating an active conversion

ADC Module Overview

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



Overview of the STM F4 ADC module [1]

The ADC model implements these logical submodules:

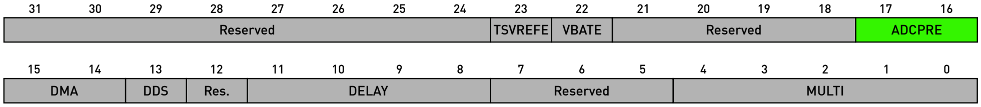
- ADC Converter with Result Registers for Injected and Regular conversion
- ADC Sample Logic for Single, Scan and Discontinuous mode
- ADC Interrupt Logic

For simplicity, the external trigger configuration shown in the figure above is neglected. The trigger to the regular and injected channels are directly accessed via the corresponding input terminals. Further, the Analog Watchdog functionalities as well as the Watchdog and DMA overrun interrupts are not part of the model. Due to simulation efficiency reasons, the ADC can not be operated in continuous conversion mode.

ADC Converter with Result Registers

The ADC module contains a converter with configurable resolution. An external voltage reference is used which can be defined in the component mask.

The period of an ADC clock, and therefore the time base for the module, is determined based on *PCLK2* and the clock divider specified in the *ADC_CCR* register.



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								TSVREFE	VBATE	Reserved				🟩 ADCPRE	

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DMA		DDS	Res.	DELAY				Reserved			MULTI				

ADC_CCER Register structure

By using the *ADCPRE* bits the ADC time base can be specified as follows:

ADCPRE[1]	ADCPRE[0]	ADC clock
0	0	PCLK2 / 2
0	1	PCLK2 / 4
1	0	PCLK2 / 8
1	1	PCLK2 / 16

The resolution of the converter can be specified with the fields *RES* of the *ADC_CR1* register given in the next section. This also influences the amount of ADC clock cycles needed for a conversion. With the *RES* bits the resolution can be specified as shown in the table below.

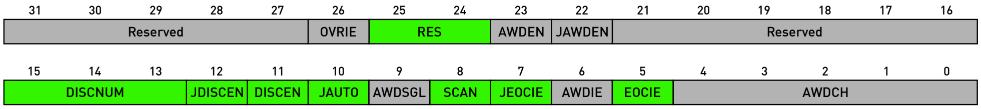
RES[1]	RES[0]	Resolution	Conversion length
0	0	12 bit	15 ADCCLK cycles
0	1	10 bit	13 ADCCLK cycles
1	0	8 bit	11 ADCCLK cycles
1	1	6 bit	9 ADCCLK cycles

For the regular channels, the hardware ADC contains a single 16-bit result register *ADC_DR*. The results of multiple, sequential regular group conversions are typically moved to the *SRAM* on the fly via the *DMA controller*. To simplify this, the *ADC_DR* terminal provides the conversion result for each of the 16 regular group members separately. For the injected channels, the *ADC_JDR* terminal provides access to the contents of all four *ADC_JDRx* registers.

The component only supports the right aligned result representation mode meaning that *ADC_CR2.ALIGN* always needs to be set to 0. In addition to this, the model provides an option to represent the conversion results as quantized double integers, which can be chosen with the mask parameter **Output Mode**.

ADC Sample Logic

The ADC model supports the single, scan and discontinuous conversion modes as well as auto-injected conversions. The continuous conversion mode is not supported due to simulation efficiency reasons. The *ADC_CR1* and *ADC_CR2* registers can be used to choose and control the used conversion mode.



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					OVRIE	RES		AWDEN	JAWDEN	Reserved					

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DISCNUM			JDISCEN	DISCEN	JAUTO	AWDSGL	SCAN	JEOCIE	AWDIE	EOCIE	AWDCH				

ADC_CR1 Register structure

The *DISCNUM* field defines the number of regular channels converted after a trigger to the regular group was received in discontinuous mode.

DISCNUM	Channels converted
000	1 channel
001	2 channels
...	...
110	7 channels
111	8 channels

The bit *JDISCEN* determines the discontinuous mode for injected channels:

- 0 - Discontinuous mode on injected channels disabled
- 1 - Discontinuous mode on injected channels enabled

With *DISCEN*, the discontinuous mode can be enabled for regular channels:

- 0 - Discontinuous mode on regular channels disabled
- 1 - Discontinuous mode on regular channels enabled

The bit *JAUTO* can be used to automatically trigger an injected group conversion after the regular group was finished:

- 0 - Automatic injected group conversion disabled
- 1 - Automatic injected group conversion enabled

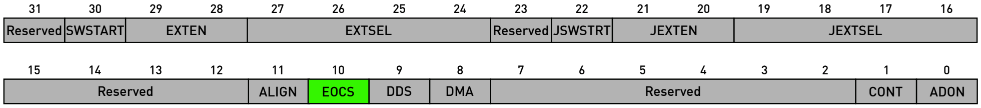
Note Be aware that *JDISCEN* and *DISCEN* exclude each other and *JAUTO* can not be used with discontinuous mode or triggers to the injected group.

With the bit *SCAN*, the user can activate the scan mode of the component allowing multiple conversion triggered by a single event.

- 0 - Scan mode disabled
- 1 - Scan mode enabled

If none of the bits *JDISCEN*,*DISCEN* and *SCAN* is set, the adc module operates in single conversion mode. The bits *JEOCIE* and *EOCIE* are further described in the interrupt section.

For more information about the different conversion modes please refer to [2].



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved	SWSTART	EXTEN		EXTSEL				Reserved	JSWSTRT	JEXTEN		JEXTSEL			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				ALIGN	EOCS	DDS	DMA	Reserved						CONT	ADON

ADC_CR2 Register structure

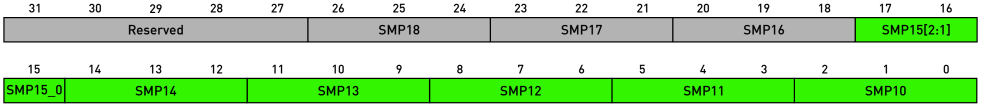
The field *EOCS* configures when the EOC flag is set while not in single conversion mode.

- 0 - EOC is set at the end of each regular group
- 1 - EOC is set at the end of each single regular conversion

---

Note The adc model assumes the adc not to operate in continuous conversion mode and to be always active. Therefore *ADC_CR2.CONT* needs to be cleared and *ADC_CR2.ADON* needs to be set while using the register-based configuration.

For every analog input, the sample time of a conversion can be configured separately using the ADC SMPRx registers.



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					SMP18			SMP17			SMP16			SMP15[2:1]	

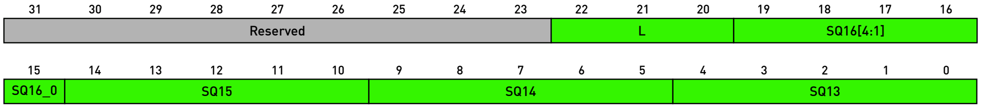
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP15_0	SMP14			SMP13			SMP12			SMP11			SMP10		

ADC_SMPRx Register structure

Note that *SMP16-SMP18* have no effect because the measurements for the temperature sensor as well as the internal reference and the battery voltage are not part of the model. For every other channel, the sampling time can be configured as follows:

SMPx	Sampling Time
000	3 cycles
001	15 cycles
010	28 cycles
011	56 cycles
100	84 cycles
101	112 cycles
110	144 cycles
111	480 cycles

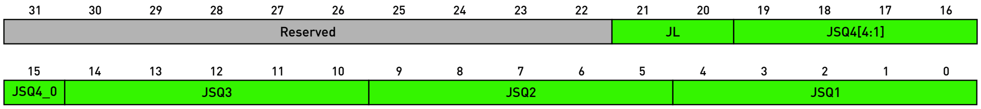
The ADC operates as a sequencer which has a maximum sequence of 16 conversion for the regular group and 4 conversions for the injected group. The input sampled by each group element as well as the sequence length can be configured via the *ADC_SQRx* and the *ADC_JSQR* registers.



ADC_SQRx Register structure

The length of the regular sequence is defined by the field *L*.

L	Sequence length	Converted elements / ADC_DR
0000	1 conversion	[SQ1]
0001	2 conversion	[SQ1 SQ2]
...	...	...
1111	16 conversions	[SQ1 SQ2 ... SQ16]



ADC_JSQR Register structure

The length of the injected sequence is defined by the field *JL*.

JL	Sequence length	Converted elements / ADC_JDR
00	1 conversion	[JSQ4]
01	2 conversion	[JSQ3 JSQ4]
...	...	...
11	4 conversions	[JSQ1 JSQ2 JSQ3 JSQ4]

After the last conversion is finished, the sequencer wraps around and restarts with the first element after the next trigger was received.

For every sequence element, the sampled input can be specified via the corresponding *SQx* or *JSQx* fields as follows:

SQx/JSQx	Input
x0000	ADC_IN0
x0001	ADC_IN1
...	...
x1111	ADC_IN15

Note The terminals *ADC_DR* and *ADC_JDR* are auto-size output terminals. This means that the width of the terminals is defined by *J* or *JL* as shown in the upper tables.

ADC Interrupt Logic

The ADC module also has a connection to the *NVIC* of the *STM F4 MCU*. The *EOC* flag is set when either the regular channel or the injected channel indicates an end of conversion. The *JEOC* flag is set when the injected group indicates a finished conversion. The fields *ADC_CR1.EOCIE* and *ADC_CR1.JEOCIE* can be used to configure the adc to provide an interrupt pulse to the corresponding output terminals.

- 0 - no interrupt pulses are generated at the EOC_INT/JEOC_INT terminal
- 1 - interrupt pulses are generated at the EOC_INT/JEOC_INT terminal

Even if there typically won't be a model of the *NVIC* within the simulation, those pulses can i.e. be used to trigger the PIL block modeling a control step triggered by a finished adc conversion.

Reference

1 - Literature Source: STM32 Reference Manual [RM0090]

5

Embedded Application

This chapter provides additional information about the STM32 F4 BLDC Current Control demo application.

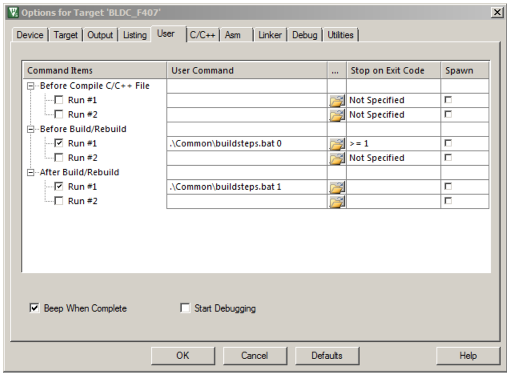
Opening the uVision Demo Project

The source code of the embedded demonstration project is provided as part of the PIL Framework installation and can be directly opened in μVision 5.5.

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



Command Items	User Command	...	Stop on Exit Code	Spawn
⊟ Before Compile C/C++ File				
☐ Run #1			Not Specified	☐
☐ Run #2			Not Specified	☐
⊟ Before Build/Rebuild				
☑ Run #1	.\Common\buildsteps.bat 0		>= 1	☐
☐ Run #2			Not Specified	☐
⊟ After Build/Rebuild				
☑ Run #1	.\Common\buildsteps.bat 1			☐
☐ Run #2				☐

☑ Beep When Complete ☐ Start Debugging

Custom build steps

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

Rebuilding the Project

The project can be compiled and flashed by clicking the corresponding symbols on the toolbar.

After reflashing the 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.

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.
- `gpio.c/h` – Initialization of inputs and outputs.

- `adc.c/h` – Initialization of adc.
- `hall.c/h` – Initialization of position measurement.
- `pwm.c/h` – Initialization/update of power stage.

Control Law

The BLDC control algorithm includes the following functionality:

1. Measurement of dc current.
2. PI current control for dc current.

The files related to the control algorithms are the following:

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

Communication Interface

The demo project either utilizes a Virtual Com Port or the serial communication interface (SCI) for exchanging information with PLECS. This can be changed by defining the symbol USARTCOMM in main.h.

- `sci.c/h` – SCI communication driver configured for VCP at the CN5 port or using the peripheral SCI module (`PB7(RX)/PB6(TX)`. Includes the communication callback function `pollSCI`.

PIL Functionality

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

- `pil.h` – PIL framework API.
- `sci.c/h` – Communication callback function. See “Communication Callbacks” (on page 33).
- `pil_ctrl.c/h` – Control callback for stepping the control tasks during a PIL simulation. See “Control Callback” (on page 38).
- `pil_symbols_p.c` – Definitions of override and read probe attributes. See “Probes” (on page 26).

- `pil_framework_fpu32.lib` – PIL framework library, compiled for `fpu32` floating point support.

IO Map

Function	**GPIO**
TIM1.CH1	PA8
TIM1.CH2	PA9
TIM1.CH3	PA10
TIM1.CH1N	PA7
TIM1.CH2N	PB0
TIM1.CH3N	PB0
ADC1 Input	PA4
Hall A-C	PC0-PC2
USART1 TX	PB6
USART1 RX	PB7
USB VCP	PA11 & PA12 (F407)
	PB14 & PB15 (F429)

Ports used for BLDC example project

electrical engineering software

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