LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY

-LIGO-

CALIFORNIA INSTITUTE OF TECHNOLOGY

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Document Type	DCC Number **T080135-v3**	October 20, 2009

AdvLigo CDS Real-time Code Generator (RCG) Application Developer's Guide

R. Bork, M. Aronsson

California Institute of Technology
LIGO Project – MS 18-34
Pasadena, CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project – NW 22-295
Cambridge, MA 01239
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

www: http://www.ligo.caltech.edu/

Table of Contents

1 Introduction

For the development of real-time controls application software, the LIGO Control and Data Systems (CDS) group has developed an automated real-time code generator (RCG). This RCG uses MATLAB Simulink as a graphical data entry tool to define the desired control algorithms. The resulting MATLAB .mdl file is then used by the RCG to produce software to run on an AdvLigo CDS front end control computer.

The software produced by the RCG includes:

- A real-time code thread, with integrated timing, data acquisition and diagnostics.
- Network interface software, using the Experimental Physics and Industrial Control System (EPICS) software and EPICS Channel Access. This software provides a remote interface into the real-time code.

2 Document Overview

This document describes the means to develop a user application using the RCG. It contains the following sections:

- Reference Section (3): The RCG produces software which integrates with various other components of CDS software. In addition, there are various files and services which must be configured prior to code operation. These items are covered under separate documentation, listed in the reference section.
- RCG Overview (4): Provides a brief description of the RCG, its components and resulting code threads.
- Application Development (5): Provides the basics for developing an application using the RCG, with a sample application file.
- Software Execution (6): Describes how to start and stop the software application.
- RCG Software Parts Library (7): Describes the various components supported by the RCG.

3 References

LIGO T080136-C CDS Software Admin Guide: Describes the various computer services and configuration files which must be in place to operate software produced by the RCG.
TBD CDS Software Development Guidelines: Provides the rules and guidelines for software development for applications which are to run in AdvLigo CDS.

4 RCG Overview

The RCG uses MATLAB Simulink as a 'drawing' tool to allow for applications to be developed via a Graphical User Interface (GUI). A basic description of this process, the RCG itself, and resulting application software is provided in the following subsections.

4.1 Code Development

Code development is done by graphically placing and connecting blocks in the MATLAB Simulink editor. The 'building blocks' supported by the RCG are included in the CDS_PARTS.mdl file. The contents of the present file are shown below, with further descriptions of the blocks listed in Section 7.

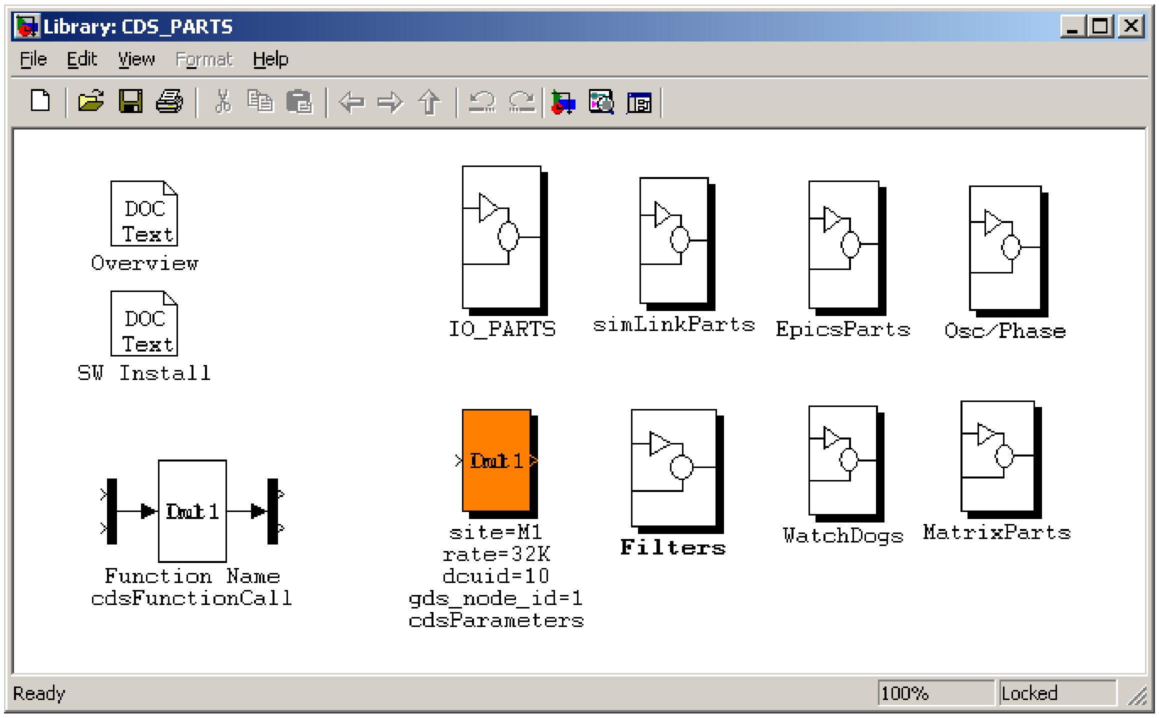


Figure 1: CDS Parts Library

Parts from the CDS library are copied (drag and drop) to the user application window and then connected to show processing/signal flow. A simple example is shown in the following figure. This example shows:

- A CDS parameter block, used to identify the desired sample rate and connection into the CDS infrastructure.
- A single, 32 channel ADC (Analog-to-Digital Converter; adc_0).
- An ADC channel selector, which is here used to pick off the first 6 ADC channels.
- A Matrix part (IN_MTRX) complete with an input Mux – multiplex or signal combiner – and an output Demux – de-multiplex or signal splitter - which routes inputs to outputs with user selectable gain for each.
- Four CDS standard IIR (Infinite Impulse Response) filter modules (FM1-4).
- A single, 16 channel DAC (Digital-to-Analog Converter).

This Simulink diagram is then saved to a user defined .mdl file, which is then processed by the RCG to provide the final real-time and supporting software which run on a CDS front end computer.

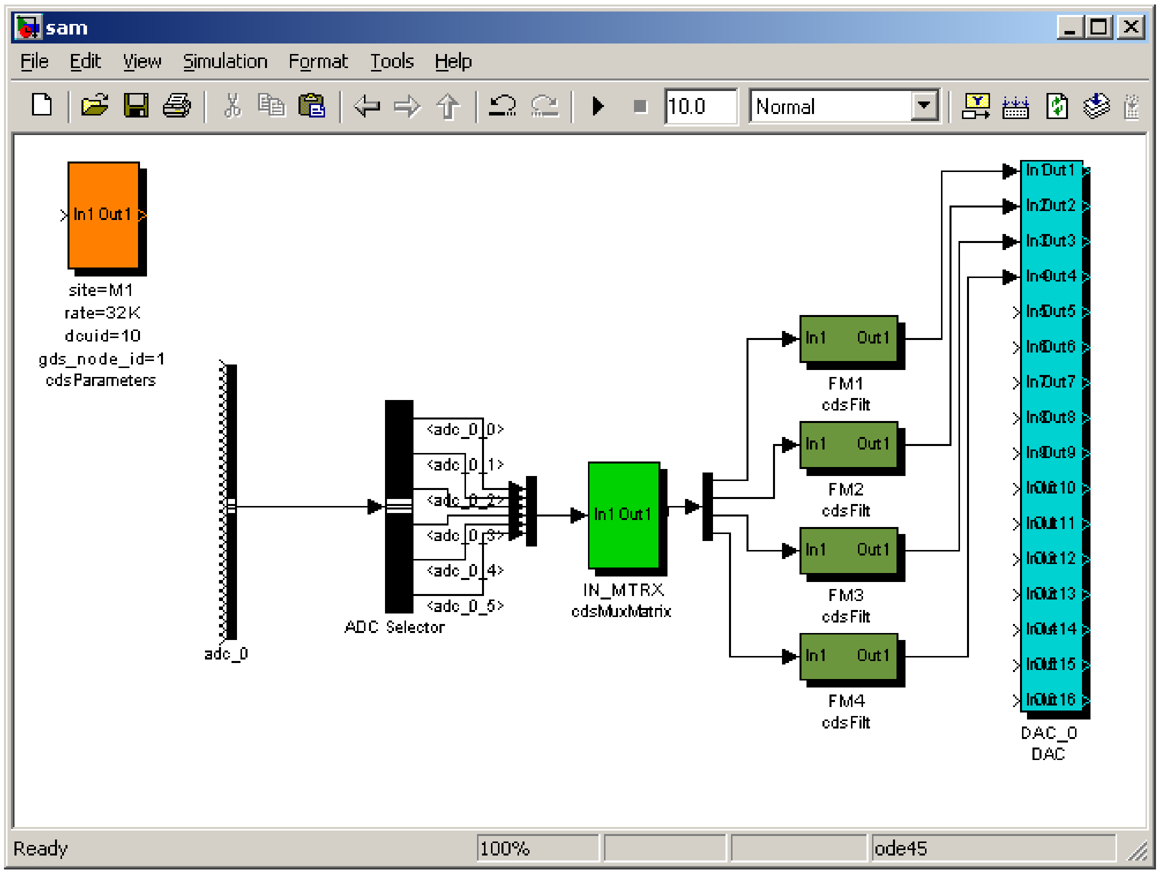


Figure 2: Sample Application

4.2 Code Generator

The code generation process is shown in the following figure and the basic process is described below.

1) Once the user application is complete, it is saved to the user .mdl file in a predefined CDS software directory.

2) The 'make' command is now invoked at the top level CDS directory. This results in the following actions:

- A CDS Perl script (feCodeGen.pl) parses the user .mdl file and creates:
    - Real-time C source code for all of the parts in the user .mdl file, in the sequence specified by the links between parts.
    - A Makefile to compile the real-time C code.
    - A text file for use by a second Perl script to generate the EPICS code.
    - An EPICS code Makefile.
    - A header file, common to both the real-time code and EPICS interface code, for the communication of data between the two during run-time.
- The compiler is invoked on the application C code file, which links in the standard CDS developed C code modules, and produces a real-time executable.

- The Perl script for EPICS code generation (fmseq.pl) is invoked, which:
  - Produces an EPICS database file.
  - Produces an executable code object, based on EPICS State Notation Language (SNL). This code module provides communication between CDS workstations on the CDS Ethernet and the real-time FE (Front End) code.
  - Produces basic EPICS MEDM (Motif Editor & Display Manager) screens.
  - An EPICS BURT (Back Up and Restore Tool) back-up file for use in saving EPICS settings.
  - The header for the CDS standard filter module coefficient file.
  - A list of all test points, for use by the GDS (Global Diagnostic System) tools.
  - A basic DAQ (Data Acquisition) file.
  - A list of all EPICS channels for use by the EDCU (EPICS Data Collection Unit).

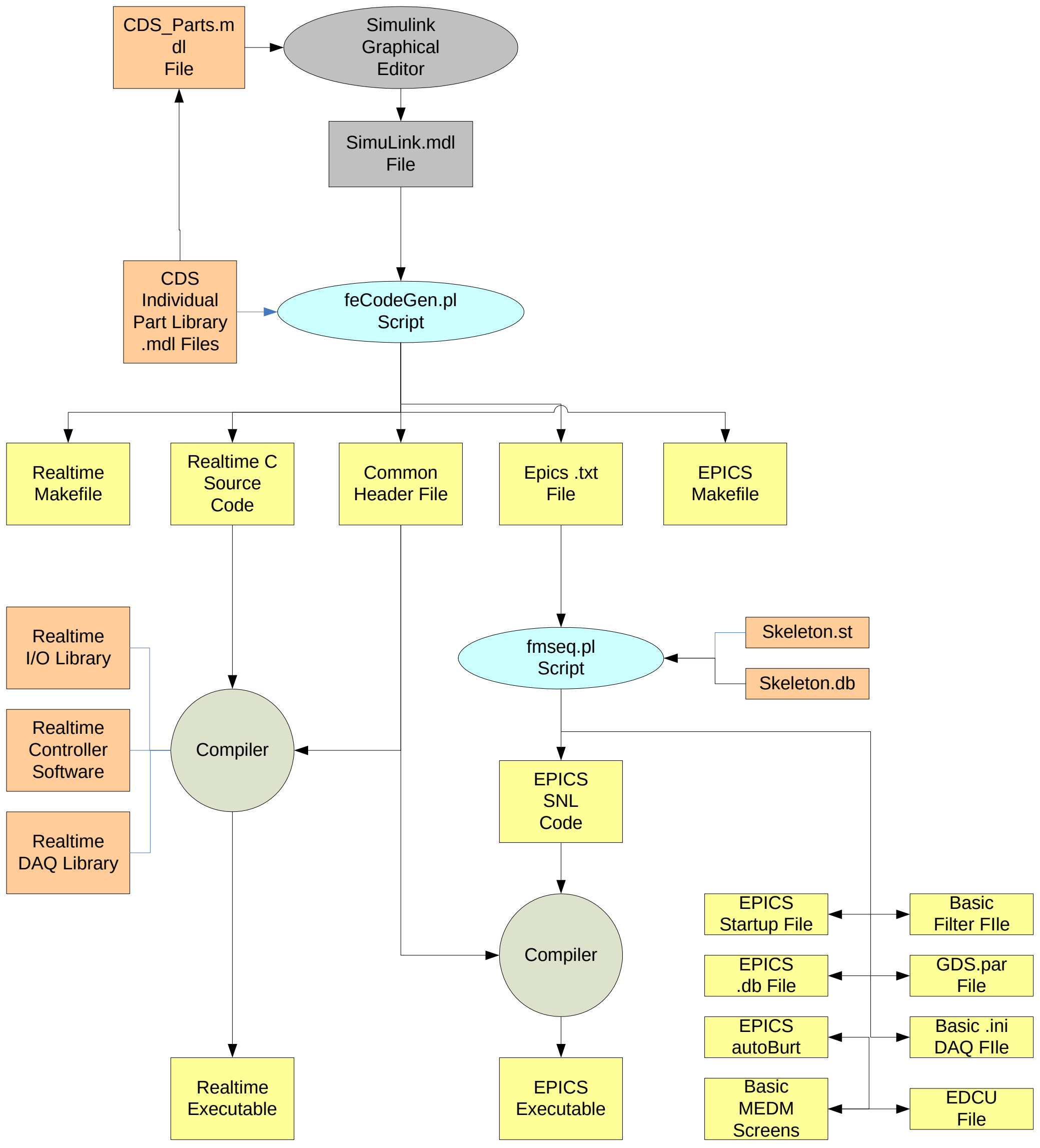


Figure 3: Code Generation

4.3 Run-time Software

The primary software modules to run on CDS FE computers are shown in the figure below. The intention is that all FE computers run the same generic code modules (highlighted in green), and that only the block labeled FE Application be specific to each FE computer.

The computer itself is to be a multi-CPU/multi-core computer, with up to 4 cores available. Generic Linux would be the operating system for the 'Non-Real-time' CPU (Central Processing Unit), and up to 3 extra available running Real-time Linux.

The 'Non-Real-time' CPU runs the following tasks:

- GDS Test Point Manager (TPM) and Arbitrary Waveform Generator (AWG). In LIGO, one TPM and one AWG was run per IFO (Interferometer) and communicated to the FE applications via Reflected Memory (RFM). In the AdvLigo scheme, the TPM/AWG runs on each FE computer and communicates to the FE application via internal memory space.
- EPICS based network interface. The purpose of this task is to relay real-time FE application information to/from EPICS operator interfaces. In LIGO, an EPICS interface task is run on a separate computer and communicates to the FE applications via RFM. In the AdvLigo scheme, there is an EPICS task on the FE computer to relay this information via the CDS network and internal computer memory.

Real-time CPUs in the FE computer run the real-time control and monitoring application. The code modules shown are inline compiled and run as a single task. The code modules that make up this task are:

- Synchronization software: This module controls initialization and timing of all other code modules. This code is slaved to the CDS timing clock used to synchronize the ADC modules.
- I/O Drivers: This code supports all input/output to the ADC/DAC modules in the I/O chassis, and data access to the CDS real-time network.
- DAQ/GDS: This module writes all data to the real-time network for data acquisition and handles all TP and AWG signals.
- FE Application: This code is specific to each FE and runs all of the necessary control algorithms, including CDS standard filter modules. To aid in the development of this software, a MATLAB Simulink tool is provided. This allows the application to be developed through a standard GUI, then compiled with the above generic modules.

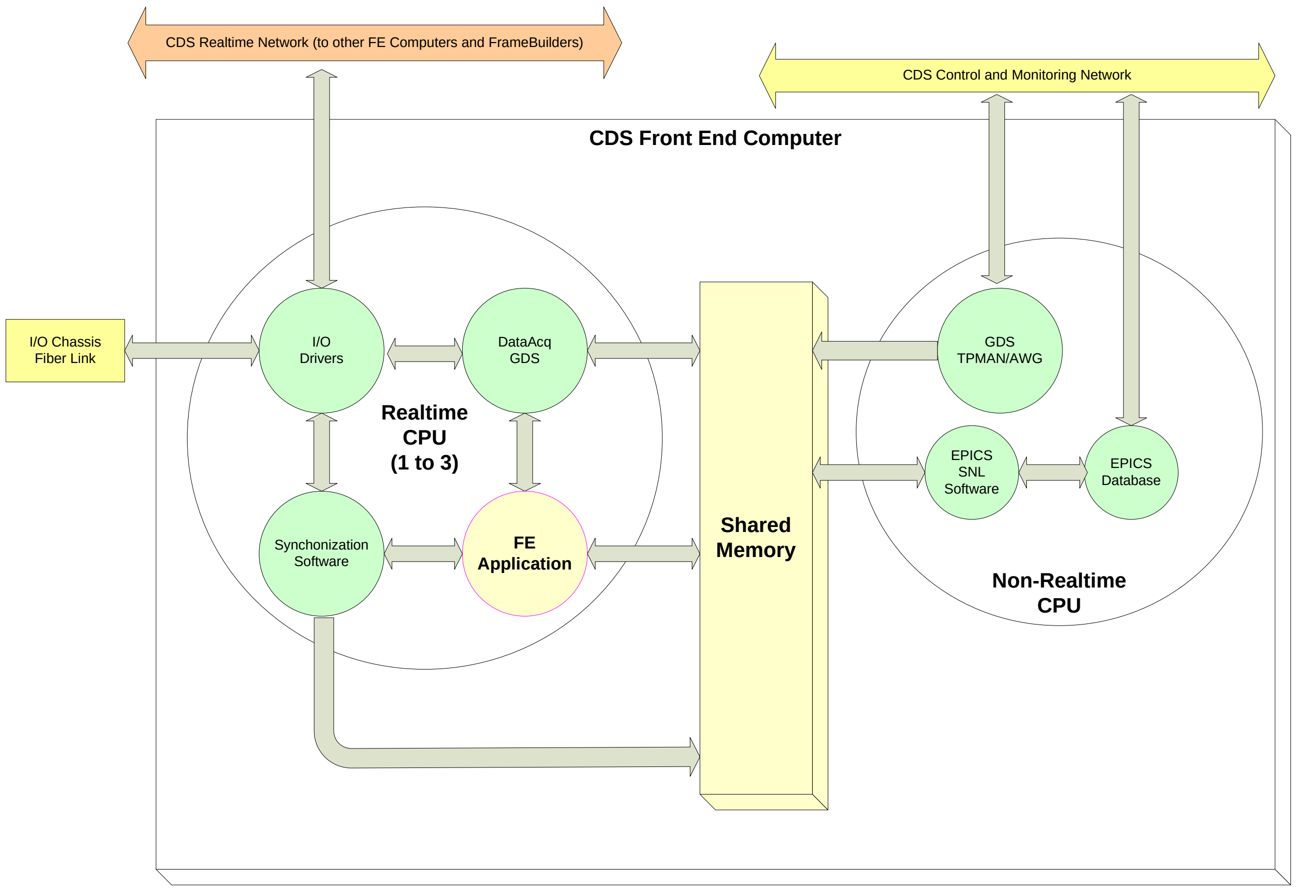


Figure 4: Runtime Software

5 RCG Application Development

This section describes how to use the RCG by stepping through a basic example.

5.1 Basic Code Development

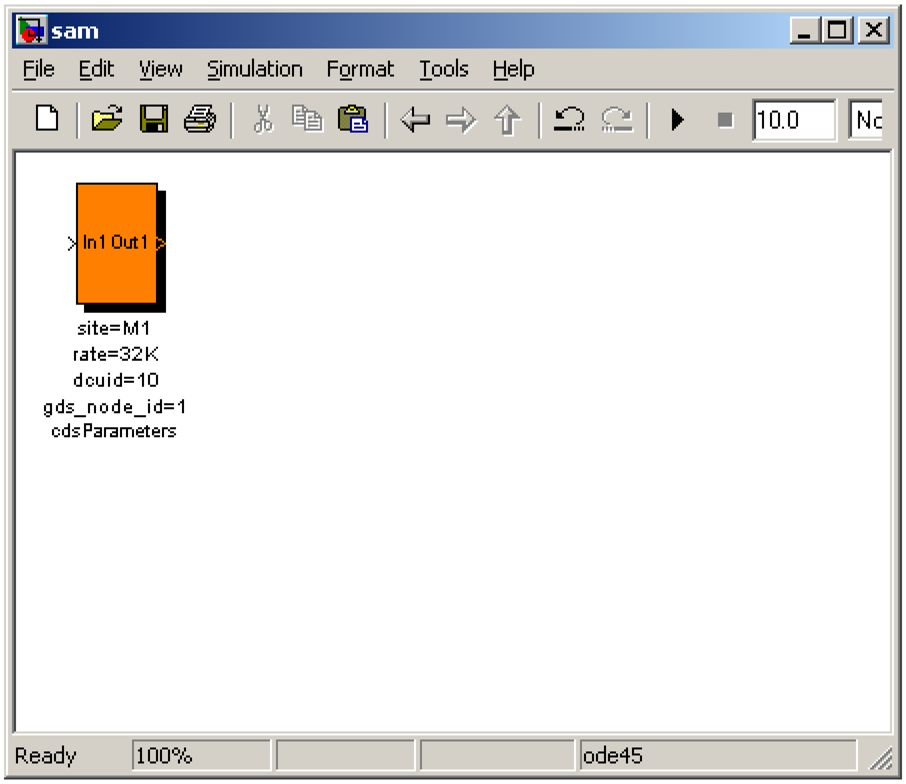
5.1.1 General Rules, Guidelines and Gotchas

Some overview notes before starting an application development process:

1) Only modules shown in the CDS_PARTS.mdl file may be used in the application development. Simulink native parts which may be used are shown in the CDS_PARTS >> simLinkParts window. A description of all available parts is given in Section 7.
2) The tool is designed to work with the LIGO CDS standard naming convention, which includes:
    a. All channel names shall be upper case.
    b. All channel names shall be of the form A1:SYS-SUBSYS_XXX_YYY where:
        i. A1 is the Interferometer (IFO) site and number, such as H1, H2, L1, M1, etc., followed by a colon (:). The IFO part of the name is set using the *cdsParameters* part in the application model (see example in next section).
        ii. SYS is a three letter system designator, such as SUS, ISI, SEI, LSC, ASC, etc., followed by a dash (-).
        iii. SUBSYS and beyond are user definable, up to a maximum channel name length of 28 characters (limit set by EPICS software). Underscores are used to further break up the name, with any number of characters in between.
3) The present release of RCG uses the first three characters of the .mdl file name, by default, as the three letter acronym for the SYS part of the channel names in the model. This naming ‘feature’ may be overridden by the use of ‘subsystem’ parts (see section 7.3.2).
4) **ALL MODELS MUST CONTAIN AT LEAST ONE ADC PART AND TWO IIR FILTER PARTS!** This has to do with the compile scripts and shared memory setups running properly.

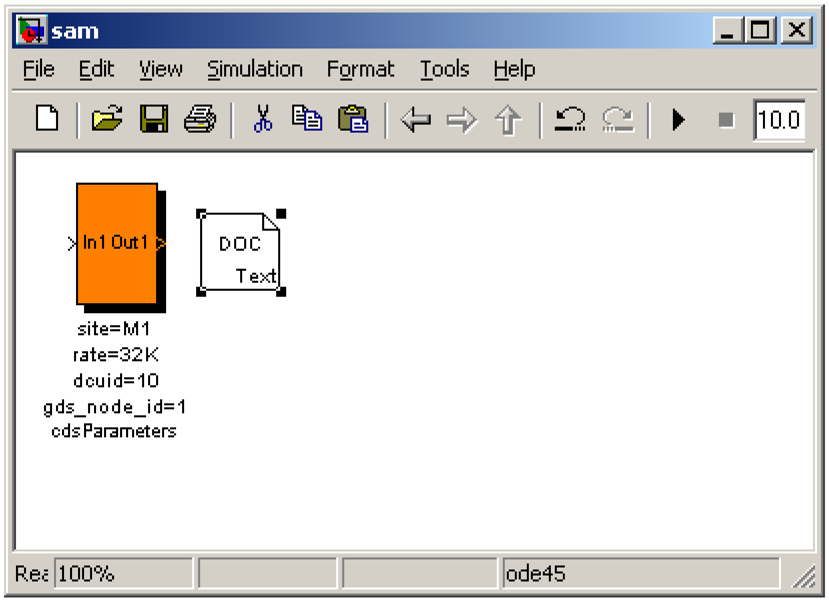
5.1.2 Example Model

1) Start MATLAB. Ensure that the *advLigo/src/epics/simLink* directory and subdirectories are in the MATLAB path.
2) Open the CDS_PARTS.mdl file. This will be used to select parts for inclusion in the user model.
3) Select File>>New>>Model from the MATLAB toolbar. This will open a new, blank Simulink window.
4) From the CDS_PARTS.mdl window, drag and drop a 'cdsParameters' block into the user model

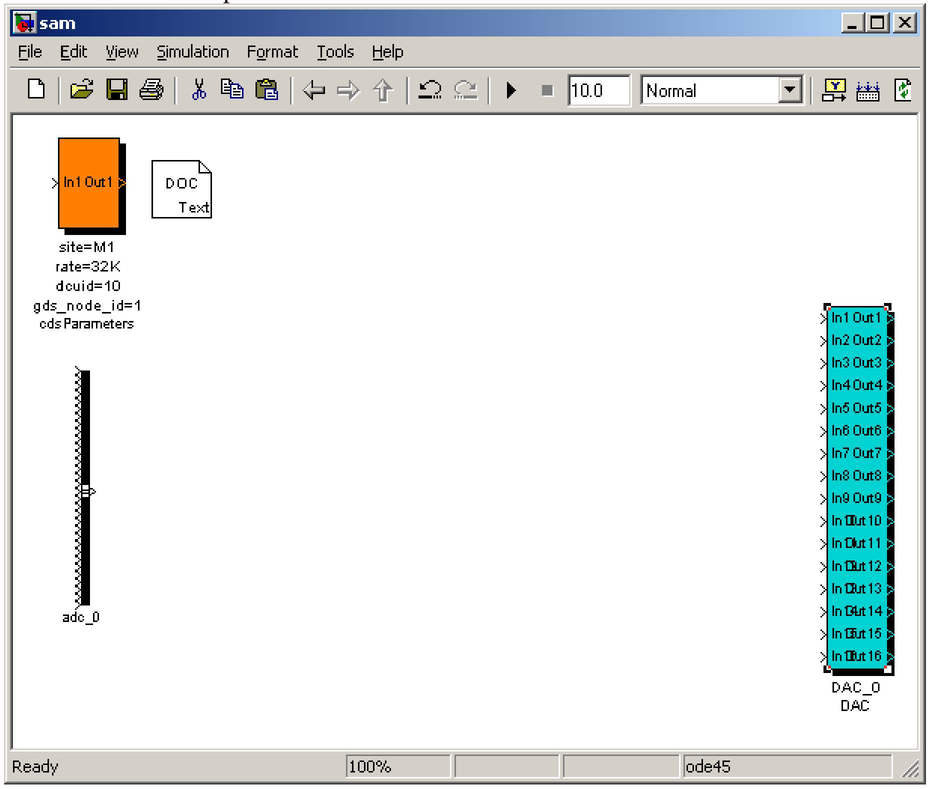


site=M1
rate=32K
dcuid=10
gds_node_id=1
cdsParameters

window. One, and only one, of these blocks is required in every user application model.
5) Define the parameters for this block by editing the text. The following is the minimal number of parameters which need to be defined. A complete list is given in section 7.1.1.
    a. site=: The RCG will name all of the parts using the LIGO CDS standard naming convention, i.e., IFO:SYS-SUBSYS_XXX_XXX_XXX to a maximum of 28 characters (EPICS limit). The IFO portion of all signal names for this model will be filled in by this site definition. In this example, M1: will be the prefix for all channel names in this model. If the code generated from this model is to run on multiple IFOs, then multiple entries can be listed after site=, e.g., site=H1,H2,L1.
    b. rate: The rate field indicates the run-time sample rate of the real-time process. Presently supported are 64K (65,536 Hz), 32K (32,768 Hz), 16K (16,384 Hz) and 2K (2,048 Hz).
    c. dcuid: Every real-time process requires a unique id. number to properly address the data acquisition system.
    d. gds_node_id: In the same manner, a unique Global Diagnostic System (GDS) id. is required for each real-time process, starting with 1 for the first model within a system. This is needed to properly attach the test point manager (TPM) and Arbitrary Waveform Generator (AWG) at run-time.
6) From the Simulink>>Model-Wide Utilities menu (i.e., click on the Simulink icon in the toolbar located second from the top in the MATLAB window, then double click on the Model_Wide



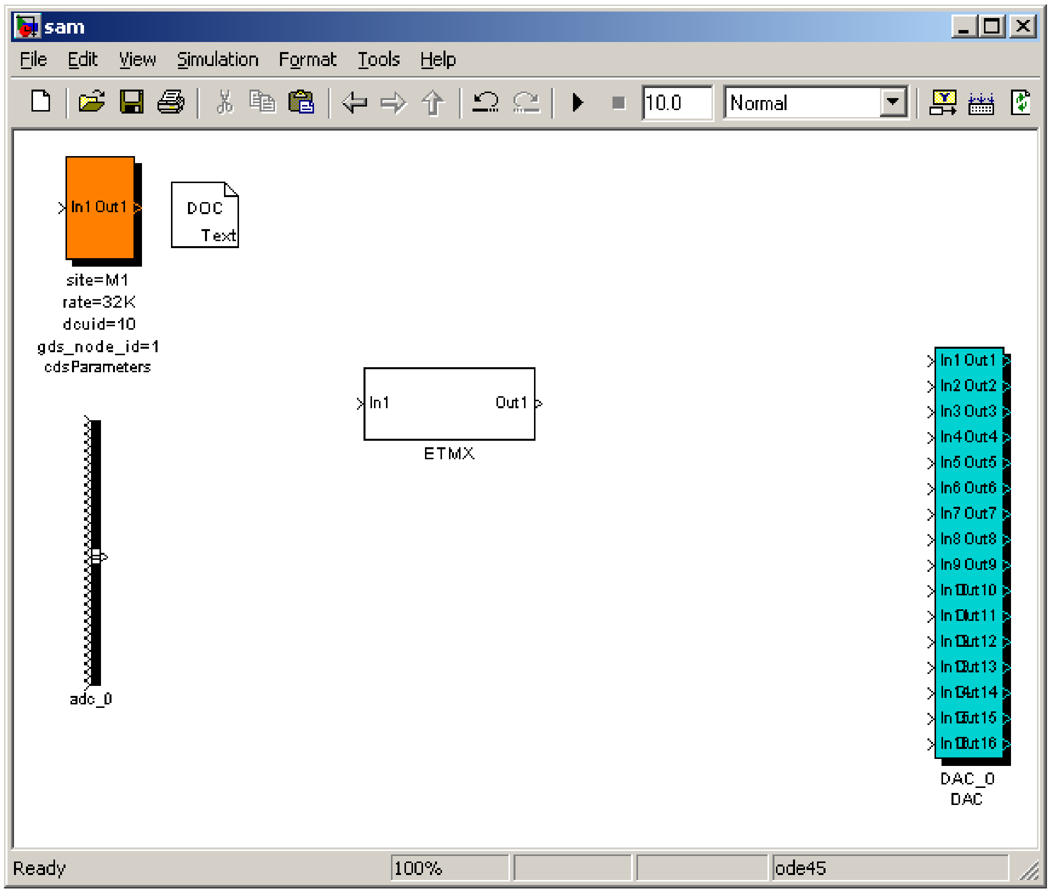
7) Add an ADC and a DAC module to the model. This is done by double clicking on the ‘I/O Parts’ block in the CDS_PARTS window, which opens the I/O parts window. Then, drag and drop the ADC and DAC parts.



8) Save this model file as ‘sam.mdl’. In the present RCG release, this must be a three letter name, as the three letters, in this case ‘sam’, are used as part of the signal names generated from this model.

9) Add a Subsystem block from the Simulink>>Commonly Used Blocks Menu (i.e., click on the Simulink icon, followed by double clicking on the Commonly Used Blocks entry in the Simulink Library Browser window, and drag and drop the Subsystem part into the user model). While a simple 'flat' model can be used, it is more common to organize the diagram using subsystems. This is done to keep the model view from becoming too complex and also allows the reuse of subsystems as 'parts'.

10) Change the name of the Subsystem part to 'ETMX'. Note that the convention is to name all parts in the model using upper case, in keeping with the CDS naming convention. In the following steps, blocks will be added to the Subsystem block. The name of every item within the subsystem block will later be prefixed by 'M1:SAM-ETMX_', where M1 came from the *cdsParameters* block, SAM comes from the name of the model file, and ETMX comes from the subsystem part name.



11) Double click on the 'ETMX' subsystem part, which will open a window showing an input connected to an output.

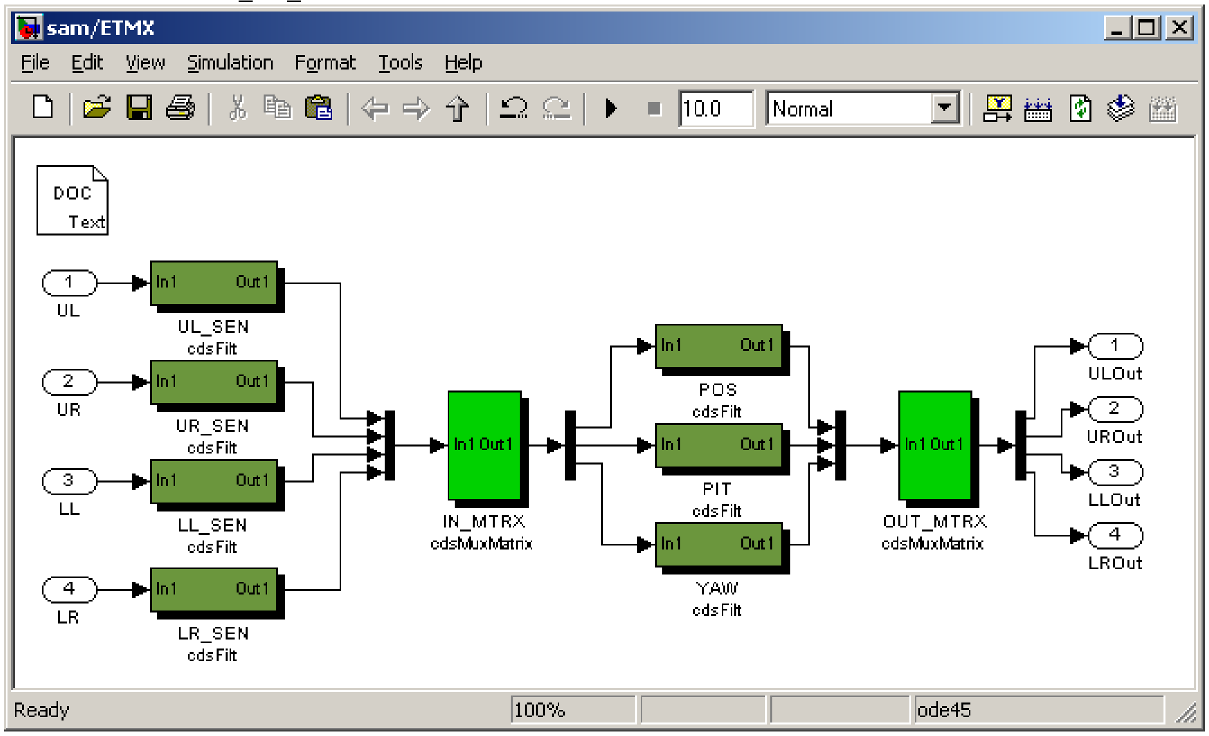
a. Disconnect the link between 'In1' and 'Out1' (i.e., click on the link between 'In1' and 'Out1', followed by clicking on the cut – scissors – icon in the MATLAB toolbar).

b. Copy 'In1' (i.e., click on 'In1', followed by clicking on the copy – double page – icon in the MATLAB toolbar, followed by clicking on the location in the user model where the copy should be placed, and finally clicking on the paste – clipboard – icon in the MATLAB toolbar) several times until there are 'In1' thru 'In4'. Do the same with 'Out1'. Change the names of these parts to something more meaningful, as these connection points will appear as part of the subsystem part at the top level diagram. In the case of this example, the four inputs are renamed UL, UR, LL, LR and the four outputs are

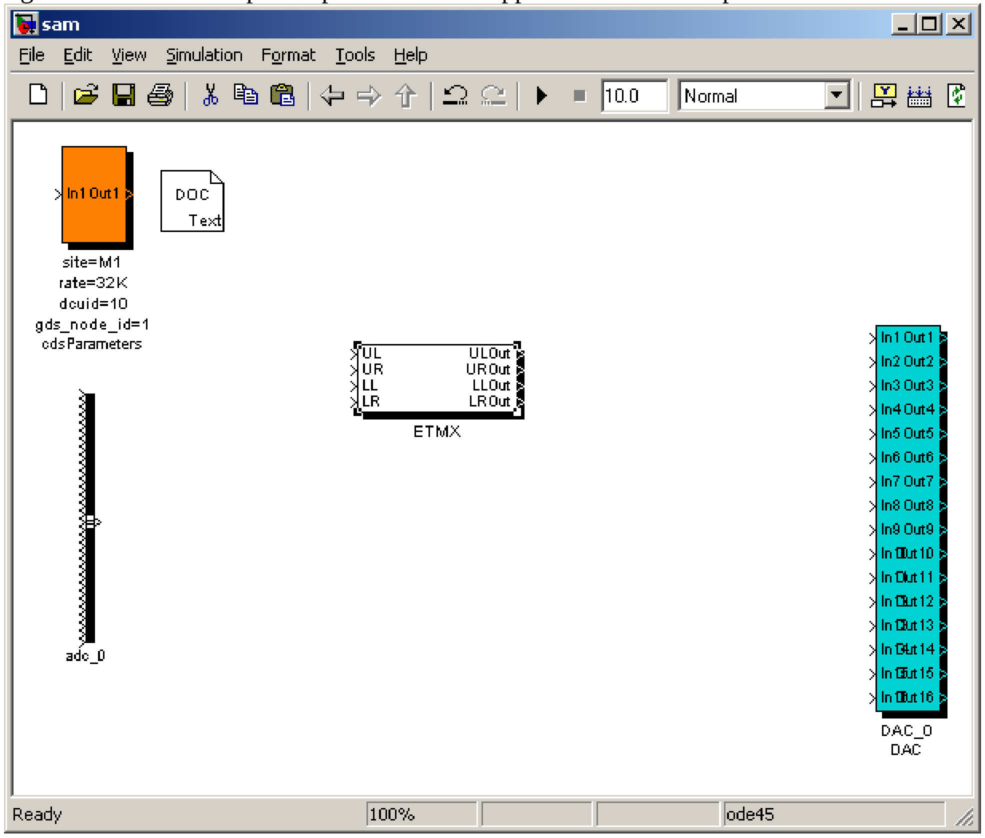
c. From the CDS_PARTS, select and place filter modules and matrix parts into the subsystem window and make connections and name changes until the window appears as shown in the next figure. Note that the number of inputs and outputs to a matrix part can be changed by double clicking on the Mux/Demux parts and entering the number of desired ports (i.e., click on the black vertical - Mux/Demux – bars connected to the cdsMuxMatrix block, which opens a window where the number of inputs/outputs can be altered).

d. In keeping with CDS standards, add a 'DOC' block in the upper left to document this code section.

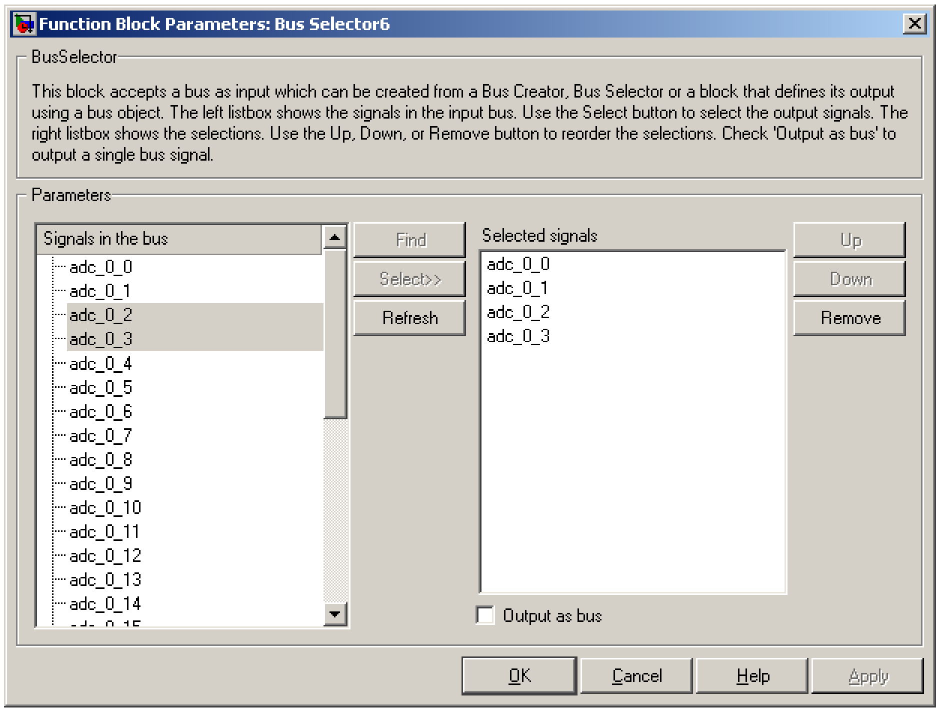
12) After step 11, the subsystem block window should look like the following figure. When the code is generated , the EPICS names of these channels will be prefixed by M1:SAM-ETMX, e.g., M1:SAM-ETMX_UL_SEN.



13) Close the subsystem window. The top level window should now appear as shown in the following figure. Note that the input/output names now appear on the ETMX part.



14) Next step is to connect ADC channel(s) to the ETMX part. From the CDS_PARTS>>IO_Parts drag and drop an ADC Selector part. Connect the adc_0 part to the ADC selector part. Double click on the ADC selector. Select any four signals as inputs from the MATLAB GUI (i.e., highlight the desired signals in the left – Signals in the bus – window and click on "Select>>", which should lead to the desired signals appearing in the right – Selected signals – window; finally click on the "OK" button).


Function Block Parameters: Bus Selector6

BusSelector

This block accepts a bus as input which can be created from a Bus Creator, Bus Selector or a block that defines its output using a bus object. The left listbox shows the signals in the input bus. Use the Select button to select the output signals. The right listbox shows the selections. Use the Up, Down, or Remove button to reorder the selections. Check 'Output as bus' to output a single bus signal.

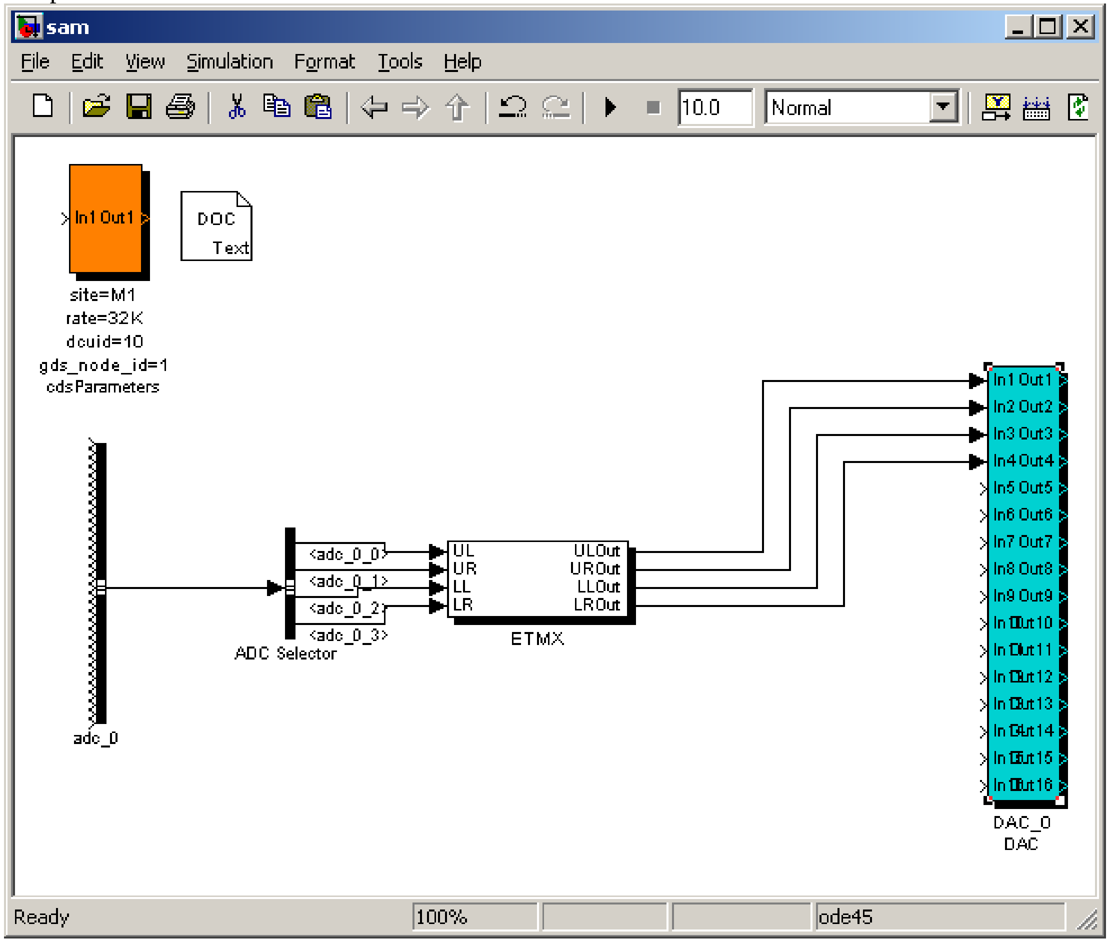
Parameters

Signals in the bus: adc_0_0, adc_0_1, adc_0_2, adc_0_3, adc_0_4, adc_0_5, adc_0_6, adc_0_7, adc_0_8, adc_0_9, adc_0_10, adc_0_11, adc_0_12, adc_0_13, adc_0_14

Selected signals: adc_0_0, adc_0_1, adc_0_2, adc_0_3

Output as bus [ ]

15) Connect the ADC selector to the ETMX part and ETMX part to the DAC and this sample model is complete.



5.2 Code Compilation and Installation

The software may be compiled in any user area that includes the *cds/advLigo* source code tree from the CDS CVS software repository. This space must be mounted to a computer which has RT Linux installed, as all compilation must be done on a real-time computer.

To compile the code:

1) Place the MATLAB .mdl file in the directory *advLigo/src/epics/simLink*
2) Move to the the *advLigo* directory.
3) Type ‘make <sys>’, where <sys> is the three letter name of the .mdl file. This command will result in the compilation of all the code, including EPICS.

Once the code is compiled, a few more commands need to be run from the *advLigo* directory to install the code for execution.

1) make install-<sys> : This command installs the code in the appropriate directories for execution and makes the automated start-up commands. The EPICS code will be copied to the */cvs/cds/<site>/target/<ifo><sys>epics* directory and the front end code will be moved to the */cvs/cds/<site>/target/<ifo><sys>* directory.
2) make install-daq-<sys> : This command creates the data acquisition file in the */cvs/cds/<site>/chans/daq* directory.
3) make install-screens-<sys> : Installs automatically generated MEDM screens in the */cvs/cds/<site>/medm/<ifo>/<sys>* directory.

5.3 Defining Multiple Models For One Computer

During run-time, the RCG code requires one or more multi-core processor(s) to operate. Core 0 is reserved for standard Linux tasks and the real-time support tasks, such as EPICS. Remaining cores may be used by the real-time code threads. By default, as in the case of the example model, at run-time, the real-time code will run on CPU 1.

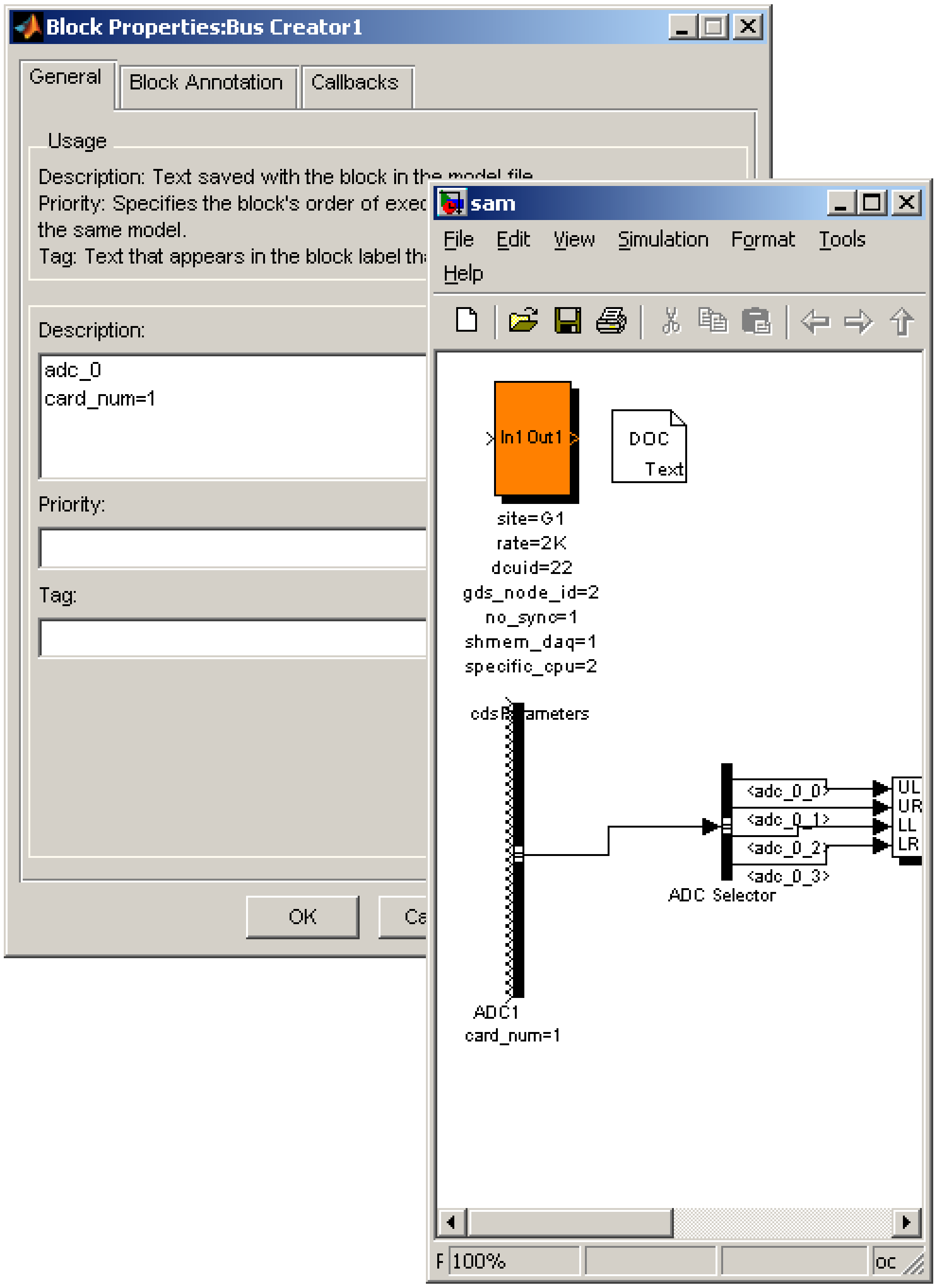
If it is desired to run multiple applications on the same computer, a couple of things need to be done:

- The support services must be configured, as described in the SysAdmin Guide.
- Applications which are destined to run on Core 2 and higher must have some additional parameters set:
    - The cdsParameter part must have specific_cpu=num, where num is the core number on which to run. This number may be 2 to 15, dependent on the number of cores on the target computer.
    - Since, in the present release, models may not share I/O cards, these cards require further definition in the model file.

Taking the previous example model as an example, to have this model run on CPU core 2 and make use of ADC card 1 (instead of the default core 1 and ADC 0 of the example model), the following changes would need to take place:

- The cdsParameter block would need to have specific_cpu=2 added.
- The adc_0 block will need card_num=1 added to the block description. This is done by right clicking on the adc_0 part and selecting Block Properties. This will bring up the following window, where card_num needs to be added to the Description field.

The Block Properties window and resulting model changes are shown below. Note that even though adc_num has been set to 1, the user application still needs to use ADC 0 and adc_0 signals for its first ADC.


Block Properties:Bus Creator1

General | Block Annotation | Callbacks

Usage

Description: Text saved with the block in the model file.
Priority: Specifies the block's order of exec
the same model.
Tag: Text that appears in the block label th

Description:
adc_0
card_num=1

Priority:

Tag:

OK | Ca

sam

site=G1
rate=2K
dcuid=22
gds_node_id=2
no_sync=1
shmem_daq=1
specific_cpu=2

6 Running the RCG Application

6.1 Loading and Executing the software

When the code is compiled and installed, it is ready to run, as outlined below. However, for data acquisition and global diagnostics to function with this software, certain parameters must be set up for these services to work properly. See the RCG SysAdmin Guide for instructions on how these parameters are set.

6.1.1 Automatic Scripts

During the make install process, scripts are generated in the */cvs/cds/<site>/scripts* area for conveniently starting and stopping the user application. This directory should be put into the user's PATH. Note that the user must have super user privileges, as the real-time code needs to be inserted into the kernel.

To start the RCG processes, type 'start<sys>', where <sys> is the name of the model file. This will result in:

- The EPICS code being started, along with an automatic restoration of the last EPICS settings (if EPICS Back Up and Restore Tool (BURT) is in the user's path and a back-up had been made previously).
- The awgtpman process will be executed to provide GDS support for this system. Note again that this task will only function properly if the appropriate system parameters have been set up, as described in the SysAdmin Guide.
- The real-time code thread will be executed and inserted into the kernel of CPU 1.

To verify that the software is functioning, use the auto generated MEDM screen, described below in section 6.2.1. There are also log files produced in the target areas for the EPICS and real-time code which provide additional diagnostic information.

To stop the software, execute the *kill<sys>* script, where again *<sys>* is the model name. This will kill all tasks associated with this model.

6.1.2 Manual Code Execution

The EPICS and real-time code may also be executed manually from the command line. This is typically only done when trying to diagnose problems or the real-time code modifications do not affect the EPICS code, such as modifications to user supplied C code modules, and it is not desired to constantly stop and start the EPICS side.

During the make install-sys process, two target directories are built in */cvs/cds/<site>/target*, one for the EPICS components (named <site><sys>epics) and one for the real-time code (named <site><sys>). EPICS and the real-time code may be started independently by using the start-up command (named startup<SITE> and startup.cmd, respectively – please note the upper case <SITE> in the former) in those directories. Note that EPICS must be running prior to starting the real-time code.

6.2 Auto Generated MEDM Screens

During the make process, various EPICS displays are automatically generated. These are fairly simple displays, to get the user started and to provide for quick testing and some quick 'copy-paste' points to use in building custom operator displays. After the make install-screens-<sys> command is executed, these displays will appear in the */cvs/cds/<site>/medm/<ifo>/<sys>* directory.

These displays are:

- <IFO><SYS>_GDS_TP.adl: Provides basic diagnostic information for the running application.
- <IFO><SYS>_ADC_X: Provides a display of all ADC input channels for quick signal checkout. Note that, in the present release, this display will only show ADC channels which are directly connected to filter modules or EPICS outputs in the model file.
- Filter module displays: For every filter module in the model file, a generic filter module display is generated.
- Matrix displays: For every matrix defined in the model, an associated EPICS display is generated.

These various displays are further described in the following subsections.

6.2.1 GDS_TP Display

A basic system diagnostic display is built for each system during the build process, with an example shown below. This display includes the following:

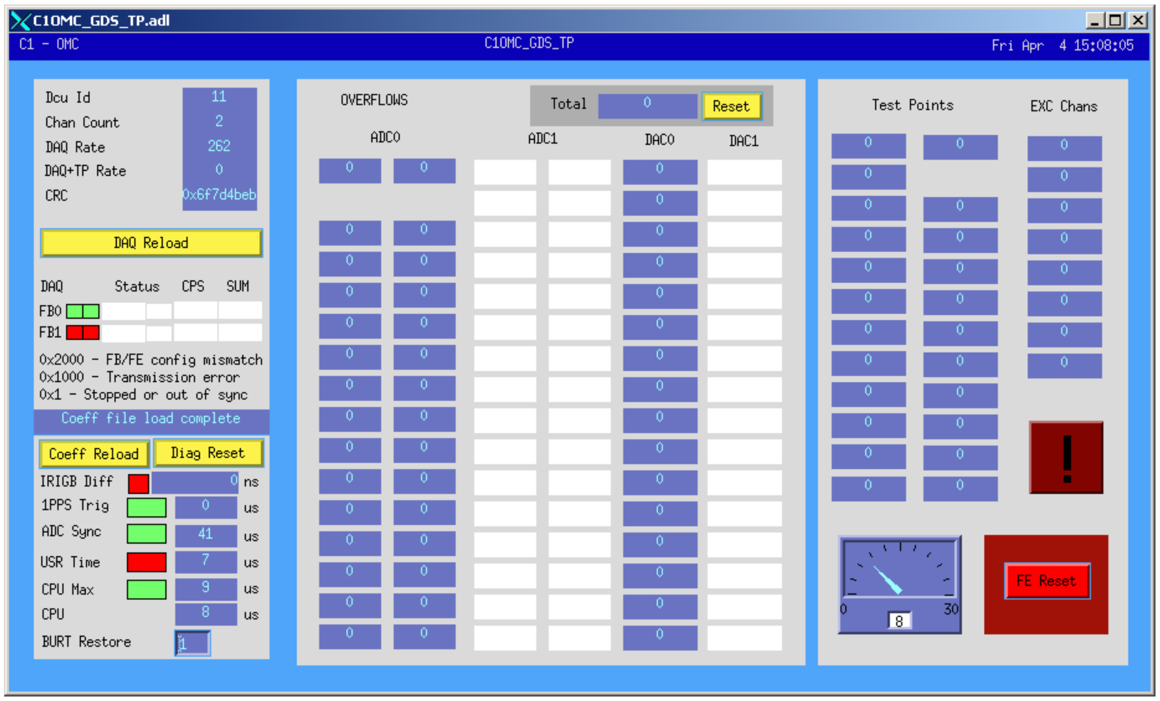
Upper Left: DAQ data and status

- Dcu Id: The DAQ node id. for this system. Each real-time process has a unique and separate id. number on the network, as defined by the MATLAB model.
- Chan Count: Number of channels presently being recorded by the DAQ system, as defined by the user in the system .ini file.
- DAQ Rate: Total data rate in Kbyte/sec for configured DAQ channels.
- DAQ + TP rate: Total data rate in Kbyte/sec being transferred by this process to the framebuilders, which is a combination of DAQ channels and selected test points.
- CRC: This is the CRC checksum, calculated from the .ini file. This number is checked by both the framebuilder and the real-time front end to verify that they have read the same .ini file.
- DAQ Reload button: When pressed, causes the real-time front end to reload the DAQ .ini file. This is to be asserted whenever a new DAQ configuration has been set by the user. Note that the framebuilder must also be reset at this time for DAQ configuration to be computed.
- Framebuilder status info: The next sub block contains framebuilder status information, as it pertains to this system. In the LIGO system, two framebuilders run on the network for redundancy, but only one framebuilder is required. The fields shown beside each framebuilder are:
    - Status block, with two red/green indicators. The left-most indicator is front end status and right-most is framebuilder status for this system.
    - Status: A hex status number, with meaning given below this block.
    - CPS: Transmission errors per second. The framebuilder performs CRC checksums on all data received from the front end system. The number in this field should be zero, but if there are continuous errors, the count will be indicated here and in the following field.
    - SUM: The total number of transmission CRC errors since the framebuilder counter was reset.

Lower Left: Front end real-time process status:

- Coeff Reload button: Pressing this button will cause the front end to reload all filter coefficients listed in its coefficient file.
- Diag Reset: Causes the reset of diagnostic values, including the CPU Max Time.
- IRIGB Diff:
- 1PPS Trig:

- ADC Sync:
- USR Time: The maximum time, in µsec, that it takes to cycle through the user application, which was developed using the RCG.
- CPU Max: Maximum amount of time, in µsec, that it took to run through a single cycle of the software, including the user application and overhead, such as I/O and DAQ, of the front end code. This field is held to the highest value until reset using the Diag Reset button.
- CPU: Similar to above, but this field is the maximum time during the last one second period.
- BURT Restore: When the software is started, the real-time code will wait for restoration of user set-point values before running. This is typically done through a BURT restore. However, this can be overridden by entering a '1' into this field.



Dcu Id 11
Chan Count 2
DAQ Rate 262
DAQ+TP Rate 0
CRC 0x6f7d4beb

DAQ Status CPS SUM
FB0
FB1

0x2000 - FB/FE config mismatch
0x1000 - Transmission error
0x1 - Stopped or out of sync
Coeff file load complete

IRIGB Diff 0 ns
1PPS Trig 0 us
ADC Sync 41 us
USR Time 7 us
CPU Max 9 us
CPU 8 us
BURT Restore 1

Total 0

Center Section: The real-time code continuously checks for ADC and DAC overflows, i.e., greater than 32,000 counts or less than -32,000 counts. If these values are exceeded, the real-time code will clamp the value to +/- 32,000 and report the error via overflow counters.

- Total and Reset (top): This field reports the total number of overflows detected for all channels. This is a running count, which may be reset using the Reset button.
- Below each ADC and DAC on this display are individual overflow counters for each channel. These fields indicate the number of overflows detected per second to help identify which channel(s) is/are having problems.

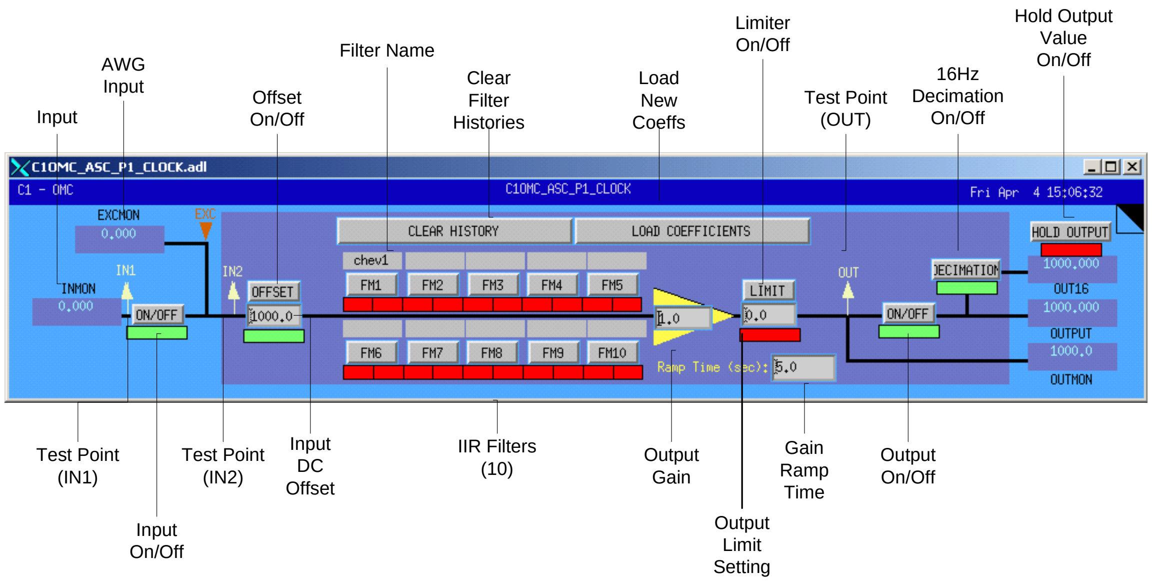
Right hand section: This section provides a list of those GDS test-point and excitation channels which are presently selected. There is also a meter representation of the maximum CPU time, same as the value in the CPU field at the lower left of this display. The meter limit is set by the sample rate of this system. For example, the system shown was set to run at 32KS/sec, so a single code cycle must complete in under 30 µsec to function properly. For 2KS/sec systems, the max time on the meter would be 480 µsec and 60 µsec for a 16KS/sec system.

6.2.2 ADC Input Display

6.2.3 Standard Filter Module Display

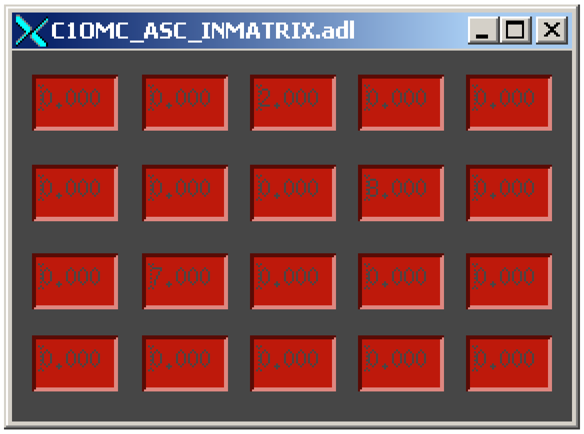
For each IIR filter module defined in the user model, a standard MEDM screen will be produced as part of the build process. An example screen is shown below. This screen contains the following EPICS I/O:

- INMON and Input On/Off: Displays the filter module input value. The following on/off switch applies/removes the input signal from the filter bank.
- EXCMON: The value of an excitation input. This field is typically 0.0 except when a GDS excitation signal is being applied.
- OFFSET value and Offset On/Off switch: Allows the user to add a DC offset to the input prior to entering the filter bank. The indicator below the offset value will be green if turned on and red if turned off.
- Filter module names and selections: The 10 available filters per bank appear to the right of the offset value field. Names, as defined using the *foton* tool, appear above each filter selection button. The filter selection buttons are used to turn the filters on/off. Below each filter button are two status indicator block. The left box indicates if a filter has been selected to be turned on (green) or off (red). The right box indicates when the real-time code has actually turned on (green) the filter or turned off (red) the filter.
- Gain and Ramping: The signal out from the filter bank may be multiplied by the gain setting. To avoid a sudden excursion of the signal when a new gain is selected, this gain may be ramped over the number of seconds entered into the Ramp Time setting. This ramping is performed by the real-time code. When the real-time code gain is not the same as the entered gain, i.e., during the ramping, the background of the triangle surrounding the gain setting will be yellow. Once the ramping is complete, the triangle will become black.
- LIMIT setting and on/off switch: The output of the filter bank may be limited by the user by setting the limit field and turning the limit switch on (green indicator). The real-time code will then limit the output to +/- the limit setting.
- Output On/Off and OUTPUT monitor: Turns the output on/off, with the filter bank output value displayed in the OUTPUT field. Note that the OUTMON (output test-point) will still have the output of the filter bank.
- DECIMATION On/Off switch and OUT16 field: The real-time code decimates the filter bank output to 16Hz, the resulting value being placed in the OUT16 field.
- HOLD OUTPUT: When selected, the output of the filter module is held to the present value (seldom used).
- CLEAR HISTORY: When selected, clears the history of all filters within the filter module. This is typically used when integrators have been defined and have rung up to a large value.
- LOAD COEFFICIENTS: Loads new filter coefficients and reloads existing filter coefficients for this filter module.



6.2.4 Matrix Display

For each matrix defined in a model, a matrix screen is automatically generated, as in the following example screen. By default, matrix elements which are set to 0.0 have their backgrounds set to red. Any other value results in a green background.



0.000	0.000	2.000	0.000	0.000
0.000	0.000	0.000	8.000	0.000
0.000	7.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000

6.3 Additional Run Time Tools

Along with EPICS MEDM, various additional tools are available to support real-time applications during run-time. These are listed below, with a few described briefly in the following subsections. For more detailed information, see the appropriate user guides for these applications.

- EPICS Back Up and Restore Tool (BURT): Used to save and restore operator settings.
- EPICS StripTool: Provides strip charting for EPICS channels.
- Dataviewer: Allows users to view DAQ and GDS TP channels, either live or from disk.

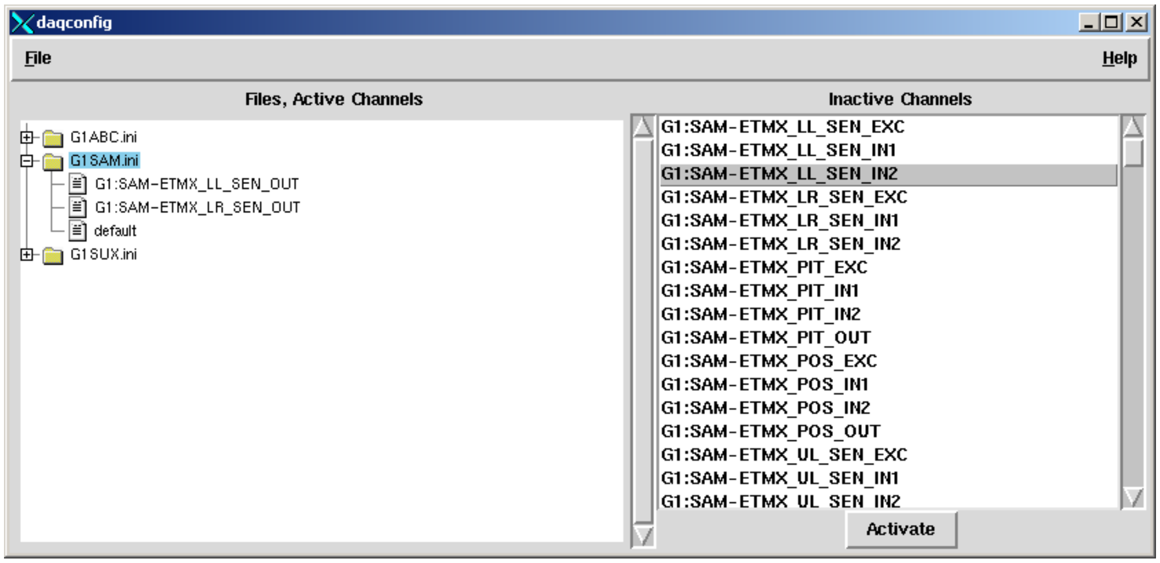
- ligoDV: Based on the GEO developed tool, this is a MATLAB tool for reading, plotting and analyzing DAQ data.
- Diagnostic Test Tool (DTT): Allows for analysis of live or recorded DAQ/TP data, particularly useful for calculating and plotting transfer functions.
- DaqGui: A graphical user interface for setting up DAQ channels.
- Foton: A GUI for the development of filter coefficients for use by the real-time software.
- Ezca based scripting tools, along with TDS scripting tools. These tools allow for the addition of automated scripts which may be used to sequence through operator settings automatically.

6.3.1 DAQ GUI

Screen shots of the DAQ configuration GUI are shown below. This tool is used to configure channels which are to be stored by the DAQ system. By default, all filter module input and output test-points are available to be recorded, but must be selected from the list and set to be stored to disk, if desired.

After the make install-daq-<sys> command is executed during the build phase, a DAQ file with all available channels is built in the */cvs/cds/<site>/chans/daq* directory (with suffix .ini). In addition, a *daqconfig* script is generated in */cvs/cds/<site>/scripts* to attach this file to the DAQ GUI. Running this script will bring up the following window, with a list of all .ini files in the *daq* directory. ***Note that this GUI is only used to configure 'fast' data channels, that is, channels which may be recorded at up to the sample rate set for that system. Slow (EPICS) channels may also be stored to disk at 16Hz, but must be separately configured, as described in section 6.4.2 below***.

Running the script will bring up the following display. This display will list all systems which have .ini files in the *daq* directory. Systems and active DAQ channels are shown in the left half of the window. A list of available channels is shown to the right.



G1ABC.ini
G1SAM.ini
G1:SAM-ETMX_LL_SEN_OUT
G1:SAM-ETMX_LR_SEN_OUT
default
G1SUX.ini

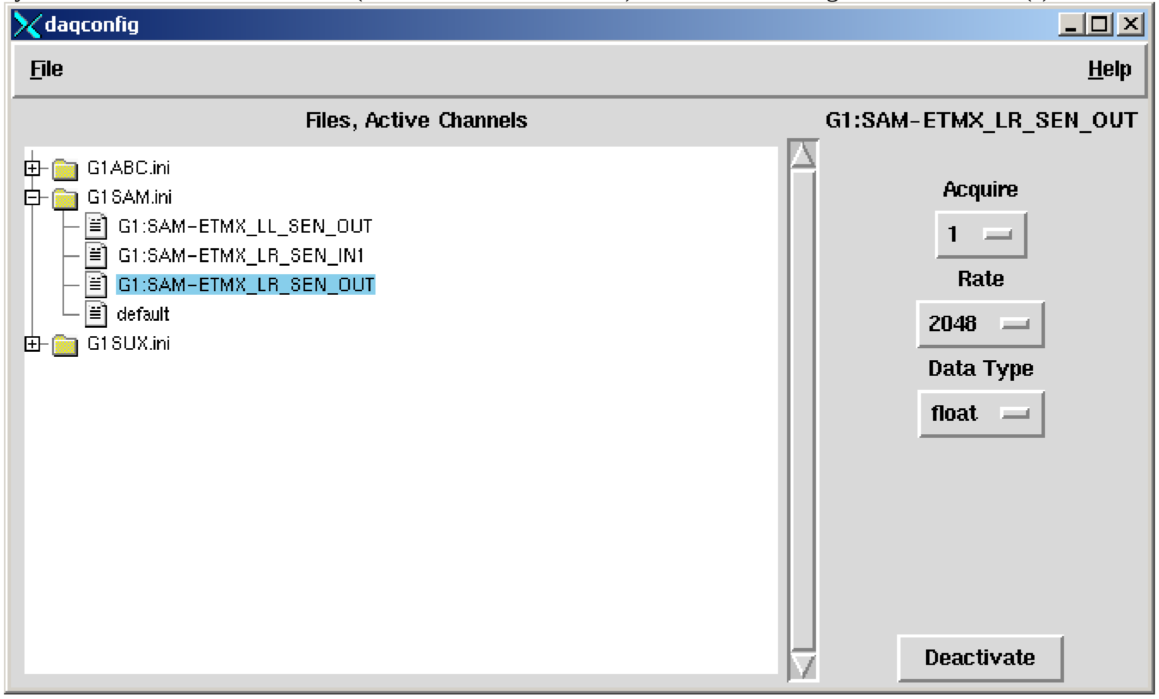
G1:SAM-ETMX_LL_SEN_EXC
G1:SAM-ETMX_LL_SEN_IN1
G1:SAM-ETMX_LL_SEN_IN2
G1:SAM-ETMX_LR_SEN_EXC
G1:SAM-ETMX_LR_SEN_IN1
G1:SAM-ETMX_LR_SEN_IN2
G1:SAM-ETMX_PIT_EXC
G1:SAM-ETMX_PIT_IN1
G1:SAM-ETMX_PIT_IN2
G1:SAM-ETMX_PIT_OUT
G1:SAM-ETMX_POS_EXC
G1:SAM-ETMX_POS_IN1
G1:SAM-ETMX_POS_IN2
G1:SAM-ETMX_POS_OUT
G1:SAM-ETMX_UL_SEN_EXC
G1:SAM-ETMX_UL_SEN_IN1
G1:SAM-ETMX_UL_SEN_IN2

Double clicking on any signal name in the active or inactive list will result in the following window being opened. From the window, the following may be selected:

- Acquire (0 or 1): Setting this value to '1' will cause the channel to be continuously sent to the framebuilder at the prescribed rate and stored to disk. Setting this value to '0' will also result in the channel being continuously sent to the framebuilder, but it will not be recorded to disk.
- Rate: The data storage sample rate may be set from 256 samples/sec up to the native sample rate of the system, as defined during the RCG model build. Decimation filters in the front end code will properly down-sample the desired channels prior to sending them to the framebuilder.
- Data Type: The data type may be set to float, int, or short. Again, the real-time front end code will perform the conversion prior to transmission.
- Deactivate: This will remove a signal from the active list.

Note that after a signal has been activated as a DAQ channel, the sample storage rate replaces the last part of the channel name. For example, if the channel name is H1:SUS-ETMX_LR_SEN_IN1 and has been set to be acquired at 256 samples/sec, the resulting DAQ channel name will become H1:SUS-ETMX_LR_SEN_256.

Once all of the desired changes have been made and the new file saved, it will be necessary to load the new configuration before it will become active. This is done by pressing the DAQ Reconfig button on the system GDS_TP MEDM screen (loads real-time front end) and then restarting the framebuilder(s).



G1:SAM-ETMX_LR_SEN_OUT

- G1ABC.ini
- G1SAM.ini
  - G1:SAM-ETMX_LL_SEN_OUT
  - G1:SAM-ETMX_LR_SEN_IN1
  - G1:SAM-ETMX_LR_SEN_OUT
  - default
- G1SUX.ini

Acquire: 1

Rate: 2048

Data Type: float

6.3.2 EPICS DAQ Configuration

EPICS channels to be stored by the DAQ system are named in a single EPICS.ini file for all systems running on the same network. This file must be located in the */cvs/cds/<site>/chans/daq* directory, and added to the *master* file list (see SysAdmin Guide).

An example file is shown below. The header portion must be as shown. Individual channels to be recorded may then be added, one channel per line, with braces around each channel name.

*********** Sample File **********************

```
[default]
dcuid=4
datarate=16
gain=1.0
```

```
acquire=1
ifoid=0
datatype=4
slope=1.0
offset=0
units=NONE

#
# HEPI channels
#
[M1:SEI-BSC_HP_INMON]
[M1:SEI-BSC_HP_OUT16]
[M1:SEI-BSC_RX_INMON]
[M1:SEI-BSC_RX_OUT16]
[M1:SEI-BSC_RY_INMON]
```

7 RCG Software Parts Library

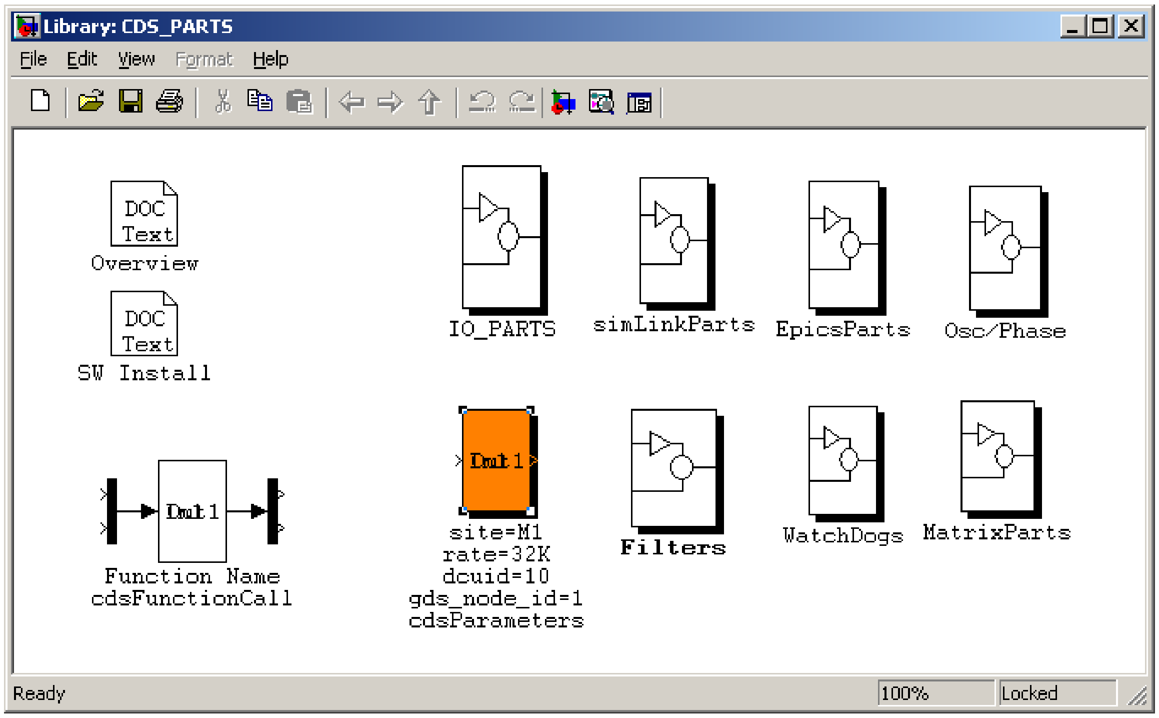
The CDS_PARTS.mdl file contains symbols for the modules supported by the RCG. Only parts defined in this library may be used with the RCG, i.e., the RCG does not support the full set of Simulink parts and some custom parts have been added for specific purposes.

7.1 Top Level Modules

CDS parts at the top level of the library include:

- cdsParameters
- cdsFunctionCall
- DOC Text/Overview
- DOC Text/SW Install

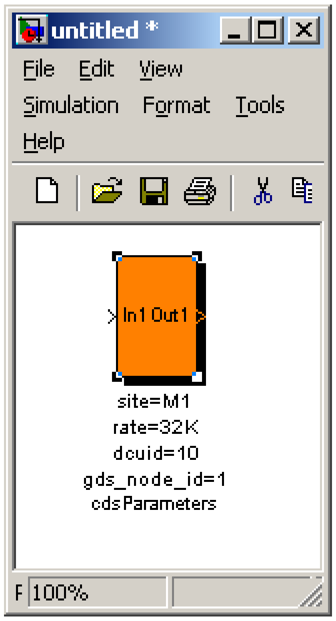
The latter two are used for documentation. Text can be entered by double clicking on one of these modules.



7.1.1 cdsParameters

7.1.1.1 Function

The purpose of this module is to define basic run-time parameters needed by the CDS RCG.



site=M1
rate=32K
dcuid=10
gds_node_id=1
cdsParameters

7.1.1.2 Usage

This module must appear once, and only once, at the top level of an RCG application model, by convention usually in the upper left-hand corner. It contains four fields which must be edited.

1) site: Somewhat of a misnomer, this field is actually the designator for the site and interferometer on which the code will run. This can be a single entry (as shown) or comma delimited for multiple IFO use, such as site=H1,H2,L1. In this case, the RCG will generate code for three IFOs. This field will be used in the EPICS channel generation as the first two characters of the channel name. In the example at right, all channel names within this RCG model will have an M1: prefix. The following sites are recognized:
    a. C (= CalTech or California Institute of Technology)
    b. G (= GEO)
    c. H (= LHO or LIGO Hanford Observatory)
    d. L (= LLO or LIGO Livingston Observatory)
    e. M (= MIT or Massachusetts Institute of Technology)
    f. S (= Stanford)
2) rate: The sample rate of the generated code must be defined as one of the supported rates:
    a. 64K (65,536 samples/sec)
    b. 32K (32,768 samples/sec)
    c. 16K (16,384 samples/sec)
    d. 2K (2,048 samples/sec)
3) dcuid: All real-time processes must have a unique (per IFO) dcuid number. This is used to identify a front end process to the data acquisition system for proper communications to the framebuilders.
4) gds_node_id: Global Diagnostic System (GDS) functions are built into every real-time application. To operate properly, each real-time application requires a unique GDS id. number.

For items 3 and 4 above, the site system administrator should be contacted for proper id. numbers if this code is to operate on an integrated CDS computer.

In addition to the above fields, there are additional optional entries. Each of these entries must be on its own line, followed by a carriage return:

- plant_name
    - Plant name.
- accum_overflow
    - ADC overflow accumulator value.
- shmem_daq
    - This results in a compiler flag such that the run-time code will use shared memory to communicate with the framebuilder software. This argument is only set if the software is to run on a standalone computer which will run the real-time code and the DAQ code.
- no_sync
    - Set if real-time code is not to be synchronized to the GPS 1PPS signal. This flag should be set if the real-time code is to be synchronized using an IRIG-B or if the system is to

- no_daq
  - System is to run without data acquisition capabilities.
- dac_internal_clocking
  - The DAC modules will be clocked using internal clock signal instead of external clock signal from timing system. This is typically only used in testing.
- no_oversampling
  - The present default is to clock all ADC/DAC at 65,536Hz, then do decimation/up-sample filtering of I/O data to match the desired servo ‘rate’. With this flag set, the decimation filtering is not performed and it is expected that the timing clock will match the ‘rate’.
- no_dac_interpolation
  - As above, except this turns off the up-sample filtering to 65,536Hz.
- compat_initial_ligo=1
  - This must be set if the computer is to run as an integrated part of initial LIGO.
- specific_cpu=x
  - Without this definition, when a model is built into an application, it will run on cpu core 1. When it is desired to run multiple real-time applications, this parameter needs to be set to the cpu core to use (2-15).
- remote_ipc_port
  - Remote IPC port value.

7.1.1.3 Operation

This component is used solely to set up appropriate compiler flags in the RCG. It is not linked as part of the real-time code.

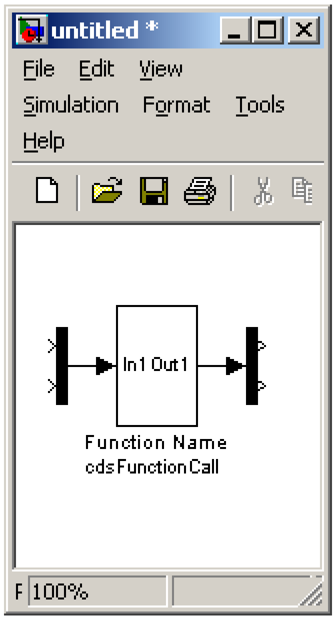
7.1.1.4 Associated EPICS Records

None.

7.1.2 cdsFunctionCall

7.1.2.1 Function

The purpose of this block is to allow users to link their own C code into the real-time application built by RCG. It is typically used when RCG does not support desired functions or the desired process is too complicated to be drawn in a model file.



7.1.2.2 Usage

Process variables are passed into and out of the user C function by connecting signals at the Mux inputs and Demux outputs. Any number of inputs or outputs may be connected by adjusting the Mux/Demux I/O sizes in MATLAB.

The ‘Function Name’ must be changed to the name of the user supplied function. Keep in mind that, as with other parts, if this part is used within ‘subsystem’ parts, it will inherit the upper level names, the same as any other part used in the .mdl file. For example, if ‘Function Name’ is re-entered as ‘prc_inv’ and this block is inside of a subsystem block named LSC, the full name of the function called in real-time will be LSC_prc_inv.

The user defined C code function must be of the form:

```
void Function_Name (double *in, int inSize, double *out, int outSize)
```

where:

- Function_Name is the full name of the function to be called. In the example above, this would be LSC_prc_inv.
- *in is a pointer to the input variables. Inputs are passed in the same order as they are connected to the input Mux.
- inSize indicates the number of parameters being passed to the function.
- *out is a pointer to the output variables. Outputs are passed back to the main code in the same order as the Demux connections.
- outSize is the number of outputs allowed from the code module.

As a simple example of user code:

```
void LSC_prc_inv(double *in, int inSize, double *out, int outSize)
{
        if (in[2] > in[0]) out[0] = in[1] * -1;
        else out[0] = in[1];
}
```

After the user code module is written, it must be placed in the appropriate directory and properly named to be compiled into the main real-time code. For example, if the above is part of a model named psl.mdl, then the code must be in the file ‘LSC_prc_inv.c’ in the *advLigo/src/fe/psl* directory.

7.1.2.3 Operation

At run-time, the code operates as defined by the user provided C code.

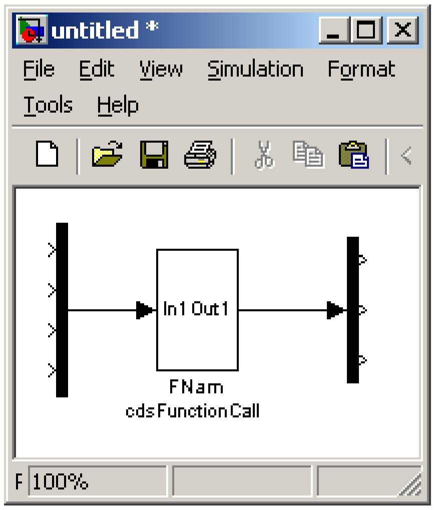
7.1.2.4 Associated EPICS Records

None.

7.1.2.5 Code Example

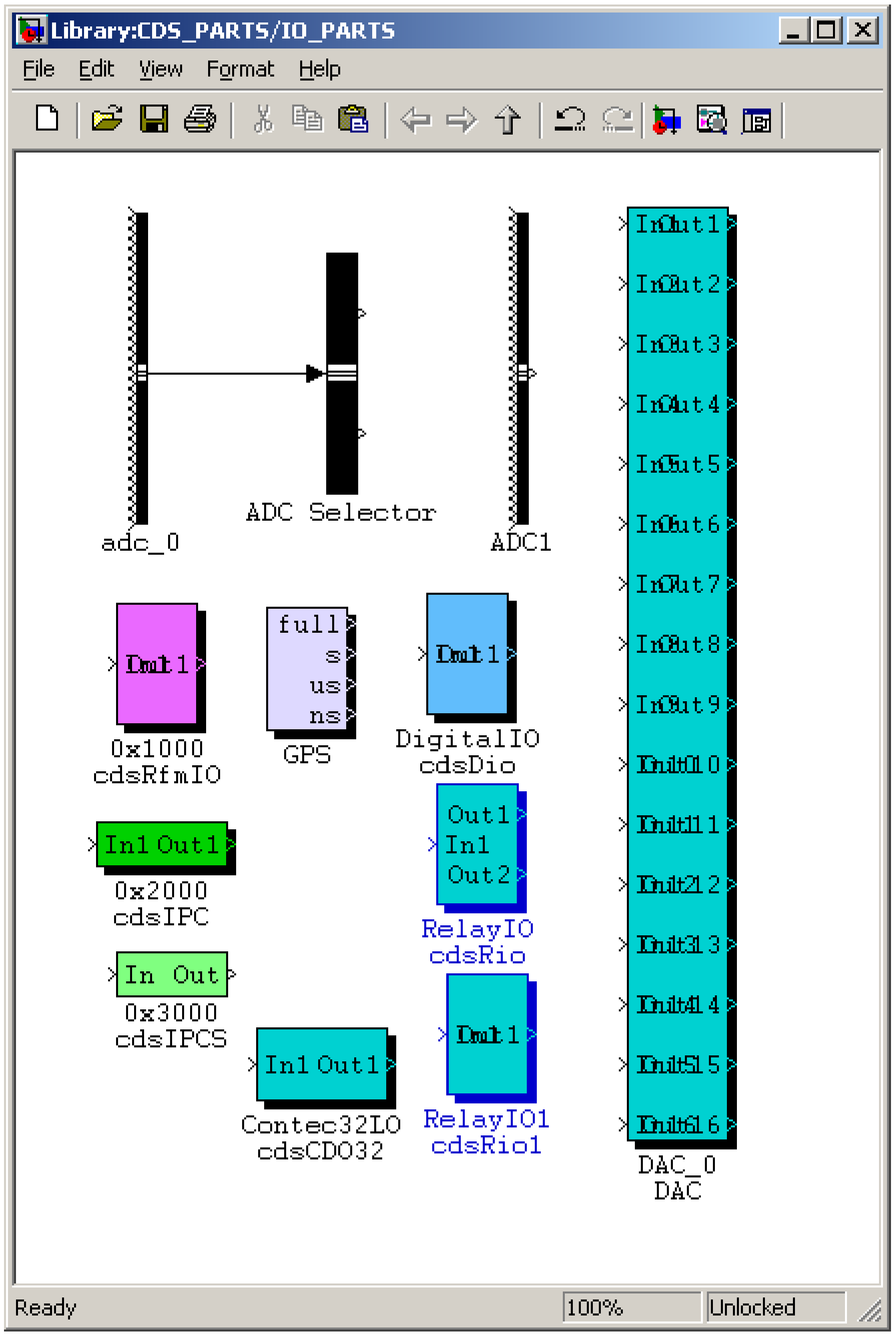
The cdsFunctionCall module generates the following C code:

```
#include "FNam.c"

double demux[3];
double mux[4];

// MUX
mux[0] = <In1[0]>;
mux[1] = <In1[1]>;
mux[2] = <In1[2]>;
mux[3] = <In1[3]>;

// Function Call
FNam(mux, 4, demux, 3);

<Out1[0]> = demux[0];
<Out1[1]> = demux[1];
<Out1[2]> = demux[2];
```

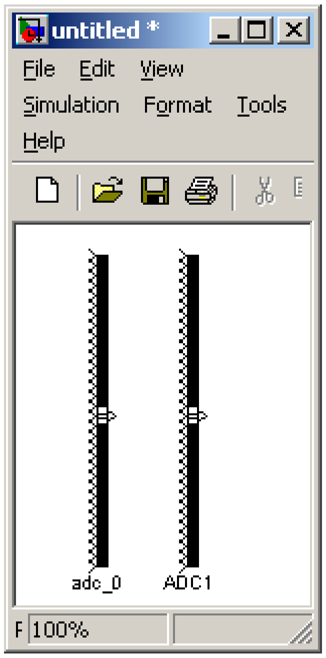


7.2 I/O Parts

The I/O parts library contains the drivers for connecting I/O modules to the system.



7.2.1 ADC



7.2.1.1 Function

The purpose of this module is to define an ADC module. At present, only the General Standards 32 channel, 16 bit ADC is supported.

7.2.1.2 Usage

Each RCG model must include at least one (1) ADC block. The output of this block must be tied to one or more ADC Selector blocks to pick out and further connect individual ADC signal channels.

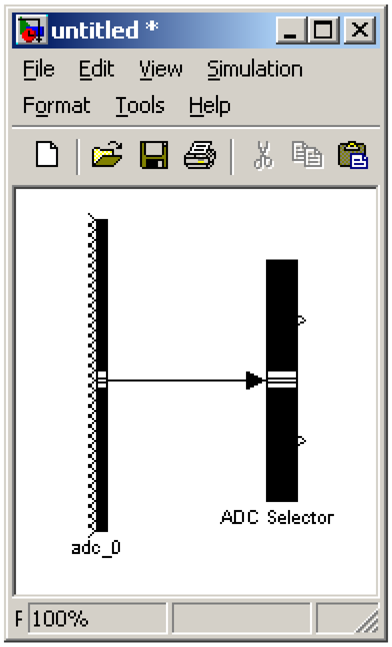
7.2.1.3 Operation

No software is directly produced for this part. Rather, it is used as an indicator of how many and of what type ADC module(s) the real-time I/O software should expect during operation.

7.2.1.4 Associated EPICS Records

None.

7.2.2 ADC Selector



7.2.2.1 Function

The function of the ADC Selector is to route selected channels from an ADC to other RCG model blocks (it is actually a Simulink Bus Selector part).

7.2.2.2 Usage

- Drag and drop the part into the model window.
- Connect the input to an ADC part.
- Double click on the ADC selector and select the desired signals from the Simulink window.
- Connect the outputs to other RCG parts.

7.2.2.3 Operation

No real-time code is directly generated to support this part. Rather, it is used by the RCG to produce appropriate signal links.

7.2.2.4 Associated EPICS Records

None.

7.2.3 DAC

7.2.3.1 Function

The purpose of this block is to allow signal connections to be output to DAC output channels.

7.2.3.2 Usage

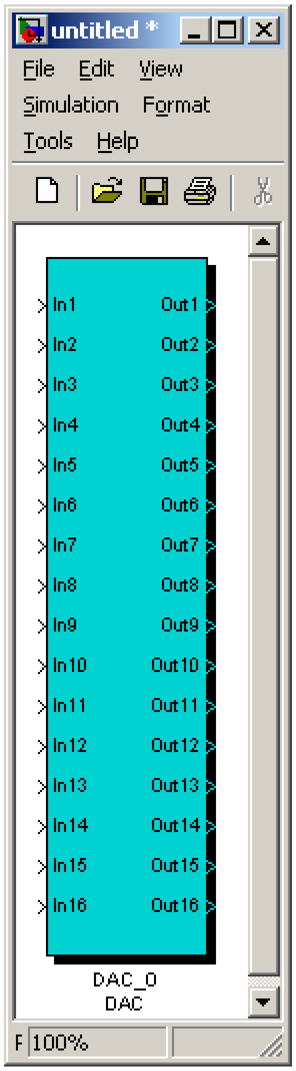
Desired output signals are connected to the 16 inputs of the DAC part. The output connections are not used.

7.2.3.3 Operation

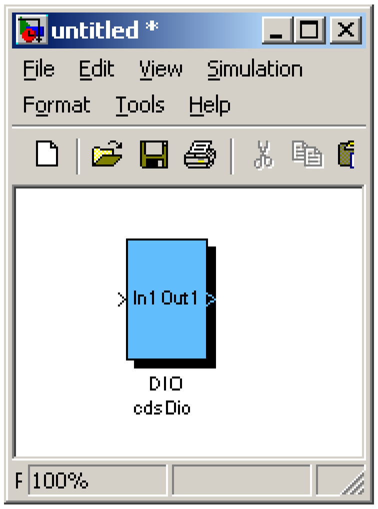
As with the ADC part, this block is only used by the real-time code to route signals to DAC modules.

7.2.3.4 Associated EPICS Records

None.



7.2.4 cdsDio



7.2.4.1 Function

Provide support for Acces 24 bit digital I/O module. The board manual can be found at PCI-DIO-24DH.PDF

7.2.4.2 Usage

In1 should be an integer, the lower 16 bits representing the bit pattern to be sent as outputs. Out1 will return an integer, the lower 8 bits of which represent the inputs to the I/O module.

7.2.4.3 Operation

The software sets the board to use 16 bits as outputs (Port A and B) and 8 bits as inputs (Port C). Software within the *advLigo/src/fe/map.c* file provides three supporting routines:

1) int mapDio(), which registers and initializes the board for use.
2) unsigned int readDio(), which is used to read the binary input bits.
3) void writeDio(), which is used to write to the 16 output bits.

Standard code definitions used in these code modules can be found in the *advLigo/src/include/drv/cdsHardware.h* file.

7.2.4.4 Associated EPICS Records

None.

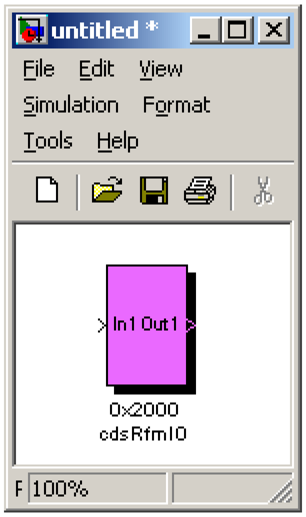
7.2.4.5 Code Example

The cdsDio module generates the following C code:

```
/* Hardware configuration */
CDS_CARDS cards_used[ ] = {
                :
        {ACS_24DIO,0},
                 :
};

// DIO number is 0
dioOutput[0] = <In1>;


<Out1> = dioInput[0];
```

(The two integer arrays dioOutput[ ] and dioInput[ ] are declared in the front-end module controller.c)

7.2.5 cdsRfmIO



7.2.5.1 Function

The RCG supports communication between computers using the GEFanuc 5565 and 5979 reflected memory modules. This block allows single signal connection to/from these modules.

7.2.5.2 Usage

If a signal value is to be sent to the module, a signal needs to be connected to ‘In1’. If a signal is to be read from a reflected memory module, then a signal should be connected from the ‘Out1’ connection. The offset from the memory board base address is entered in the block label field. In the example at right, the memory offset is set to 0x2000.

7.2.5.3 Operation

The real-time code provides a single write or read at the specified memory board offset in the form of a double precision float.

7.2.5.4 Associated EPICS Records

None.

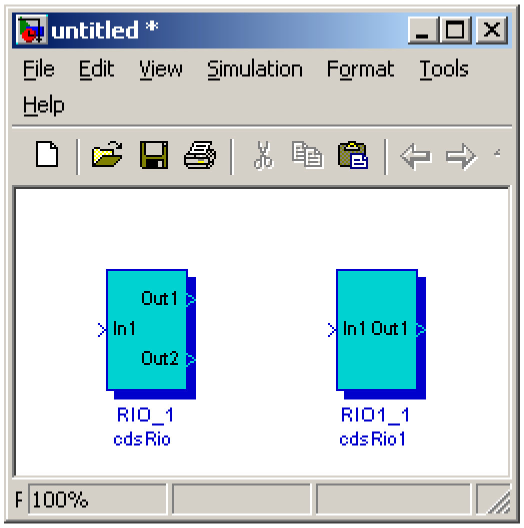
7.2.5.5 Code Example

The cdsRfmIO module generates the following C code:

```
if (cdsPciModules.pci_rfm[0] != 0) {
  // RFM output
  *((double *)(((char *)cdsPciModules.pci_rfm[0]) + 0x2000)) = <In1>;
}


<Out1> = cdsPciModules.pci_rfm[0]? *((double *)(((void *)cdsPciModules.pci_rfm[0]) + 0x2000)) : 0.0;
```

(The PCI hardware structure cdsPciModules is declared in the front-end module controller.c)

7.2.6 cdsRio and cdsRio1



7.2.6.1 Function

Provide support for Acces 8 (cdsRio part) and 16 bit relay modules (cdsRio1 part). The board manuals can be found at PCI-IIRO-8.PDF and PCI-IIRO-16.PDF.

7.2.6.2 Usage

When used, the part name must be modified to indicate the instance of the card. For example, when using an 8 bit module (cdsRio), the name of the part must be RIO_moduleNumber (RIO_0 for first instance of the module type on the bus). Same needs to be done for the 16 bit part (cdsRio1_0).

The input to both parts is an integer, the lower 8 or 16 bits representing the output bit pattern to the module.

In the case of the cdsRio part, two outputs are provided. Out1 simply returns the value written at In1. Out2 will read the 8 bits of the module input register.

Out1 of the cdsRio1 part will return an integer, the lower 16 bits of which represent the 16 input bits of the module.

7.2.6.3 Operation

Code support for these two module types is incorporated into the *advLigo/src/fe/map.c* file.

For the 8 bit module:

1) int mapIiroDio(), which registers and initializes the module for use.
2) void writeIiroDio(), which outputs the value to the I/O module.
3) unsigned int readIiroDio(), reads binary inputs from module.
4) unsigned int readIiroDioOutput(), read back the value written to the output register by the writeIiroDio() function (just a check that value was written correctly).

For the 16 bit module:

1) int mapIiroDio1(), registers and initializes the module for use.
2) void writeIiroDio1(), writes 16 bit pattern to I/O module output register.
3) unsigned int readIiroDio1(), reads the 16 bit input register.

Standard definitions used in these code modules can be found in the *advLigo/src/include/drv/cdsHardware.h* file.

7.2.6.4 Associated EPICS Records

None.

7.2.6.5 Code Examples

The cdsRio module generates the following C code:

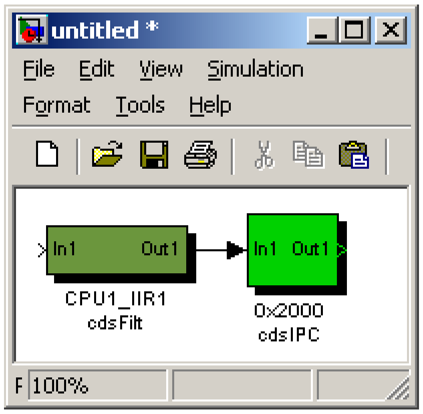
```
/* Hardware configuration */
CDS_CARDS cards_used[ ] = {
            :
    {ACS_8DIO,1},
            :
};

        :
rioReadOps[<i>] = <0, 1, or 2>;
        :

// Rio number is 1 name RIO_1
rioOutput[1] = <In1>;

        :

<Out1> = rioInputInput[1];
<Out2> = rioInputOutput[1];
```

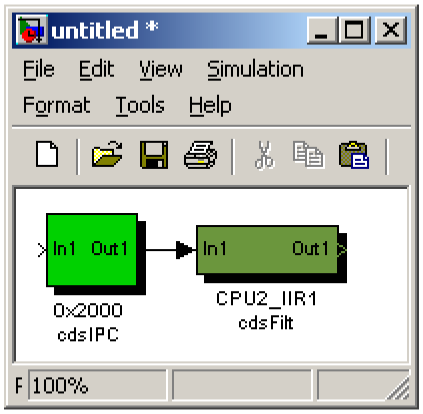
The cdsRio1 module generates the following C code:

```
/* Hardware configuration */
CDS_CARDS cards_used[ ] = {
            :
    {ACS_16DIO,1},
            :
};

// Rio1 (IIRO-16) number is 1 name RIO1_1
rioOutput1[1] = <In1>;

        :


<Out1> = rioInput1[1];
```

(The integer arrays rioReadOps[ ], rioOutput[ ], rioOutput1[ ], rioInputInput[ ], rioInputOutput[ ], and rioInput1[ ] are declared in the front-end module controller.c)

7.2.7 cdsIPC





7.2.7.1 Function

The purpose of this module is to allow communications, via shared memory, between two or more real-time processes running in the same computer, but on separate CPU cores.

7.2.7.2 Usage

The user needs to change the label to a hex value, for example 0x2000. This part needs to exist in both the ‘sender’ model and the ‘receiver’ model, with the same address in both.

7.2.7.3 Operation

If there is a signal connected at ‘In1’ (of the cdsIPC module), then this will result in the signal data being written to the address location as a double precision float. Conversely, if the ‘Out1’ is connected, data will be read in from the prescribed memory location as a double precision float. Communications at run-time use the ‘ipc’ (inter-process communication) shared memory block.

Warning:
This communication is asynchronous, i.e., the ‘receiver’ will not wait for the ‘sender’. Therefore, it is up to the user to decide and take care of any synchronization needs.

Warning:
All computer cores on the same computer will use the same ‘ipc’ shared memory block. Therefore care must be taken that models use unique addresses to communicate with each other.

7.2.7.4 Associated EPICS Records

None.

7.2.7.5 Code Examples

The cdsIPC module in the first (‘sending’) process generates the following C code:

```
double ipc_at_0x2000;

ipc_at_0x2000 = <In1>;

  // All IPC outputs
  if (_ipc_shm != 0) {
   *((double *)(((char *)_ipc_shm) + 0x2000)) = ipc_at_0x2000;
  }
```

The cdsIPC module in the second (‘receiving’) process generates the following C code:

```
double ipc_at_0x2000 = *((double *)(((void *)_ipc_shm) + 0x2000));

<Out1> = ipc_at_0x2000;
```

Or, more specifically (including the IIR Filters in the above illustration):

The cdsIPC module in the first (‘sending’) process generates the following C code:

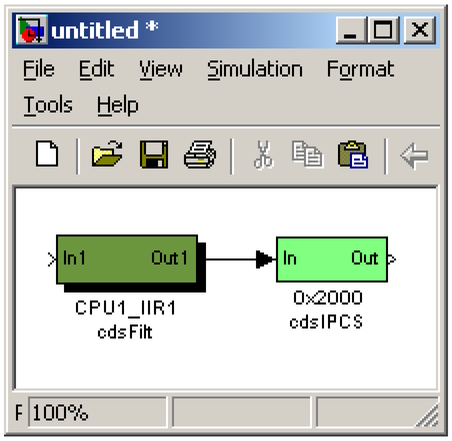
```
double ipc_at_0x2000;

ipc_at_0x2000 = cpu1_iir1;

   // All IPC outputs
   if (_ipc_shm != 0) {
     *((double *)(((char *)_ipc_shm) + 0x2000)) = ipc_at_0x2000;
   }
```

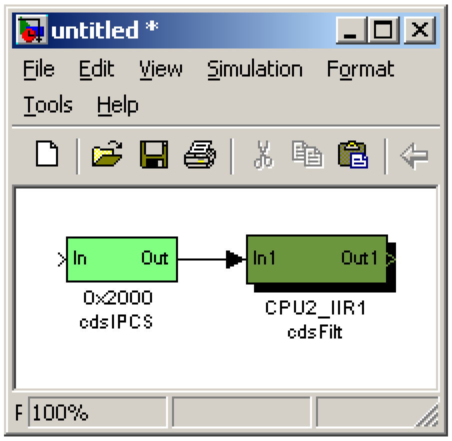
The cdsIPC module in the second (‘receiving’) process generates the following C code:

```
double ipc_at_0x2000 = *((double *)(((void *)_ipc_shm) + 0x2000));

// FILTER MODULE
cpu2_iir1 = filterModuleD(dsp_ptr,dspCoeff,CPU2_IIR1,ipc_at_0x2000,0);
```

(The pointer _ipc_shm to the inter-process communication area is declared in the front-end module controller.c)

7.2.8 cdsIPCS





7.2.8.1 Function

This part sends cycle count information between two real-time processes running on separate computer cores via shared memory. It is used to verify that the two (or more) related processes are in sync with each other.

7.2.8.2 Usage

The shared memory address must be specified in the range of 0x1000 to 0x3000 on an 8 byte boundary. One of these parts should be put in each of the two applications to be monitored, both with the same address specification. The part which is to send the cycle count should have its input connected (doesn't really matter what the input part connection is) and the receiver part should have its output connected. The output connection is typically to an EPICS OUTPUT part to view the status information (should always be zero if two applications are in sync).

7.2.8.3 Operation

During execution, the real-time code for each application maintains a "cycle counter", which continuously counts from 0 to the (user specified application rate – 1) each second. For example, if a model is specified to run at 32K, this counter increments from 0 to 32,767 every second. The send part (input connected, no output connected) will send this cycle count + 1. The receive part (output connected) will read this value from shared memory and subtract its cycle count. If the two applications are in sync, then the output of the receive part should always be zero.

7.2.8.4 Associated EPICS Records

None.

7.2.8.5 Code Examples

The cdsIPCS module in the first ('sending') process generates the following C code:

```
if (_ipc_shm != 0) {
  // IPCS output
  *((float *)(((char *)_ipc_shm) + 0x2000)) = (cycle + 1)%FE_RATE;
}
```

The cdsIPC module in the second ('receiving') process generates the following C code:

```
<Out> = _ipc_shm? *((float *)(((void *)_ipc_shm) + 0x2000)) - cycle : 0.0;
```

Or, more specifically (including the IIR Filters in the above illustration):

The cdsIPCS module in the first (‘sending’) process generates the following C code (no change):

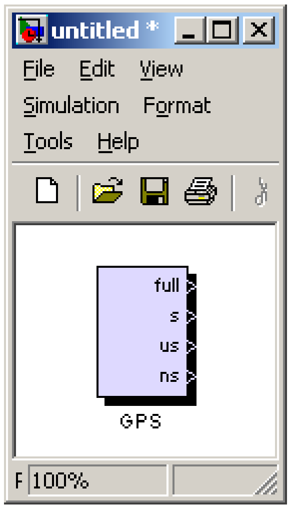
```
if (_ipc_shm != 0) {
  // IPCS output
  *((float *)(((char *)_ipc_shm) + 0x2000)) = (cycle + 1)%FE_RATE;
}
```

The cdsIPC module in the second (‘receiving’) process generates the following C code:

```
cpu2_iir1 = filterModuleD(dsp_ptr, dspCoeff,CPU2_IIR1,_ipc_shm? *((float *)(((void *)_ipc_shm) +
0x2000)) - cycle : 0.0,0);
```

(The pointer _ipc_shm to the inter-process communication area is declared in the front-end module controller.c)

7.2.9 GPS



7.2.9.1 Function

Return GPS time information from an IRIG-B interface module.

7.2.9.2 Usage

7.2.9.3 Operation

7.2.9.4 Associated EPICS Records

None.

7.2.9.5 Code Example

The GPS module generates the following C code:

<full> = cycle_gps_time;

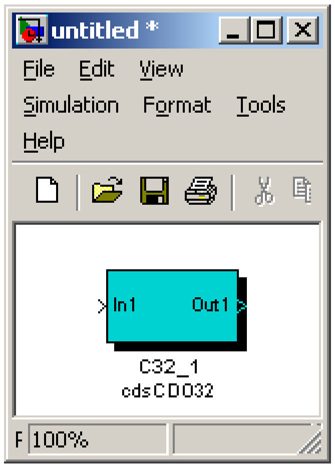
<s> = (unsigned long)cycle_gps_time;

<us> = cycle_gps_time - (unsigned long)cycle_gps_time;

<ns> = cycle_gps_ns;

(The double precision floating-point parameter cycle_gps_time and the integer parameter cycle_gps_ns are declared in the front-end module controller.c)

7.2.10 cdsCDO32



7.2.10.1 Function

This module provides I/O support for the Contec 32 bit, PCIe binary output module. The specification sheet can be found at Contec32output.pdf.

7.2.10.2 Usage

In1 should be connected to a 32 bit value to be sent to the module. Out1 will return the value from the board output register, which should be the same as the input value request.

7.2.10.3 Operation

Code support for this module can be found in the *advLigo/src/fe/map.c* file. Support routines are:

1) int mapContec32out(), register and initialize module for use.
2) unsigned int writeCDO32l(), write 32 bit value to the module output register.
3) unsigned int readCDO32l(), read the 32 bit value from the module output register (used to verify write function).

7.2.10.4 Associated EPICS Records

None.

7.2.10.5 Code Example

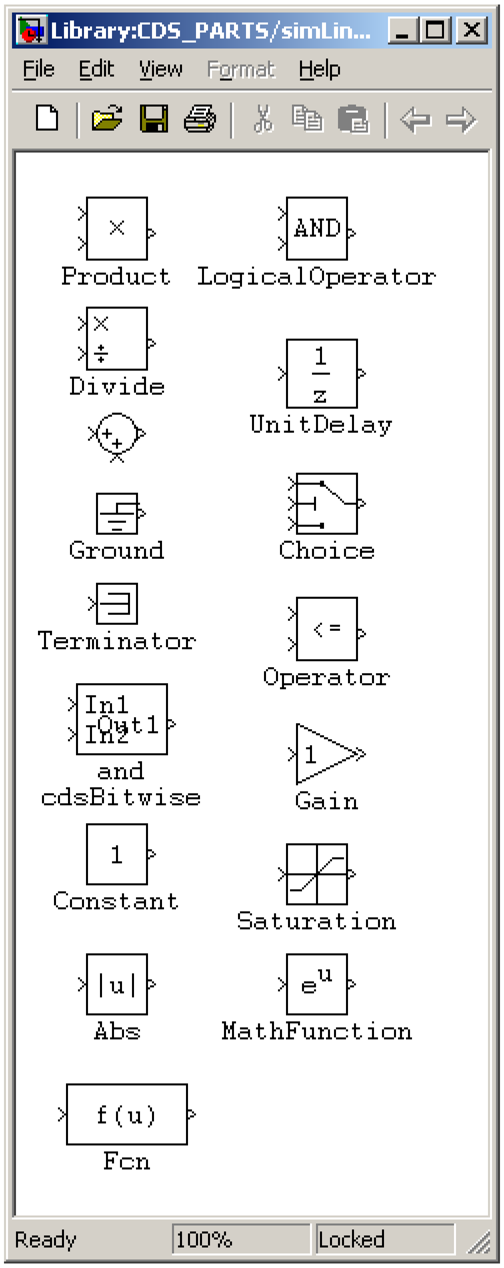
The cdsCDO32 module generates the following C code:

```
/* Hardware configuration */
CDS_CARDS cards_used[ ] = {
                :
        {CON_32DO,1},
                :
};

// CDO32 number is 1 name C32_1
CDO32Output[1] = ((int)<In1> << 16) + ((int)<In1> & 0xffff);

<Out1> = (CDO32Input[1] >> 16);
```

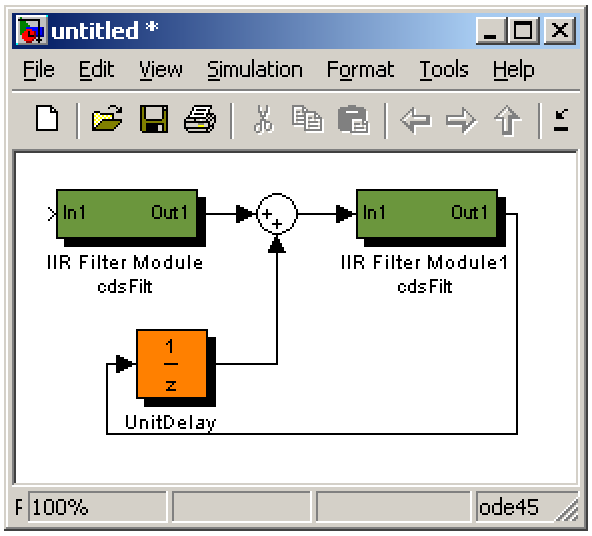
(The integer arrays CDO32Output[ ] and CDO32Input[ ] are declared in the front-end module controller.c)

7.3 Simulink Parts

The RCG supports a number of standard Simulink parts, as shown in the simLinkParts window (at right). In general, the code generated by the RCG behaves as it would in a Simulink model. Special cases are described in the following subsections.



7.3.1 Unit Delay



7.3.1.1 Function

Typically, the RCG produces sequential code that starts with ADC inputs, performs the required calculations, and ends with the DAC outputs. However, there are cases where calculations performed within the code are to be fed back as inputs on the next code cycle. In these cases, the desired feedback signal must be run through a UnitDelay block to indicate to the RCG that this signal will be used on the next cycle

7.3.1.2 Usage

An example showing the use of the UnitDelay block is shown at right. If the output of Module 1 were to be tied directly back to the summing junction at the input, it would produce an infinite loop in the code generator. By placing the UnitDelay in line, the output of Module 1 is sent back to its input on the next cycle of the software.

7.3.1.3 Operation

Introduces a one cycle delay between input and output.

7.3.1.4 Associated EPICS Records

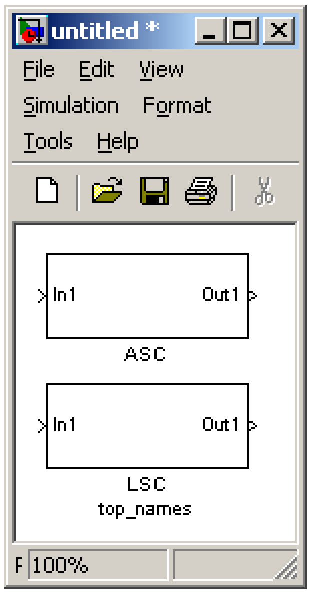
None.

7.3.1.5 Code Example

The UnitDelay module generates the following C code:

```
static double unitdelay;

<Out> = unitdelay;

// DELAY
unitdelay = <In>;
```

7.3.2 Subsystem Part



7.3.2.1 Function

This is a standard MATLAB part for grouping individual parts into a subsystem.

7.3.2.2 Usage

Any number of parts within the application model may be grouped into a subsystem using the MATLAB subsystem part. The RCG uses the assigned name as a prefix to all block names within the subsystem. This is done in two ways:

- In the top example at right, if it is at the top level of the model, all signal names for blocks within ASC would become SITE:ModelFileName-ASC_xxxx. So, if the model file name is omc.mdl and site defined as L1, names for parts within the ASC subsystem part would become L1:OMC-ASC_xxxx.
- In the lower example (LSC), a tag has been added (using the Block Properties Window) "top_names". This is a flag to the RCG to use the name of this subsystem to replace the model file name. Using the same example as above, all parts within this subsystem would be prefixed L1:LSC-xxxx.

The use of the 'top_names' subsystem part tags provides a couple of useful features:

1) A single model may contain parts with multiple SYS names in the LIGO naming convention. As seen in the example above, SYS is OMC (model name) for all ASC subsystem parts (L1:OMC-ASC_), but L1:LSC- for all LSC subsystem parts. In the same manner, ASC could also be defined 'top_names' and the results would be L1:ASC- and L1:LSC-.
2) Multiple models may contain the same SYS name. This allows models running on different processors to use the same SYS identifier in the signal names.

Warning: Since the name of all subsystems marked with the 'top_names' tag are used to replace the three character SYS part in the LIGO naming convention, this name must be 3 characters in length, no more, no less!

Warning: Subsystems with the 'top_names' tag may only appear at the highest level of the model, i.e., they may not be nested within other subsystems.

7.3.2.3 Operation

The subsystem part is only used by the RCG to produce appropriate signal names.

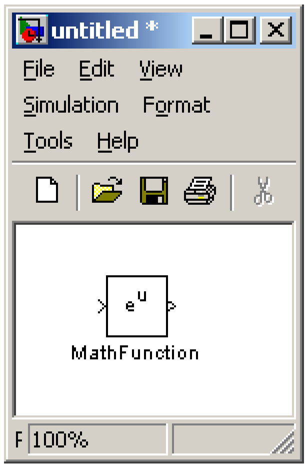
7.3.2.4 Associated EPICS Records

None.

7.3.3 MathFunction

7.3.3.1 Function

This module is used to include one of several mathematical functions in a model.



7.3.3.2 Usage

Currently, the following mathematical functions are supported:

- Square of input value.
- Square root of input value.
- Reciprocal of input value.
- Modulo of two input values.

7.3.3.3 Operation

When using this module, place it in the model window and double click on the icon. This brings up a Function Block Parameters window. Click on the down arrow at the right end of the “Function:” line. This brings up a list of mathematical functions. Click on one of the supported functions (square, sqrt, reciprocal, or mod), followed by clicking OK. Please note that clicking on any of the non-supported functions (exp, log, 10^u, log10, magnitude^2, pow, conj, hypot, rem, transpose, or hermitian) will result in a fatal error when attempting to make (compile) the model.

The square function will calculate the square of any input (double precision) value and pass it on as the output value (in double precision).

The square root function will calculate the square root of any positive (double precision) value and pass it on as the output value (in double precision). If the input value is negative or equal to zero, the output value will be set to zero.

The reciprocal function will calculate the inverse of any input (double precision) value and pass it on as the output value (in double precision), unless the input value is equal to zero in which case the output value will be set to zero.

The mod (modulo) function takes two input values, In1 and In2. Since the modulo function only operates on integer values, the output value (Out1, in double precision) is calculated as:

Out1 = (double) ((int) In1%(int) In2)

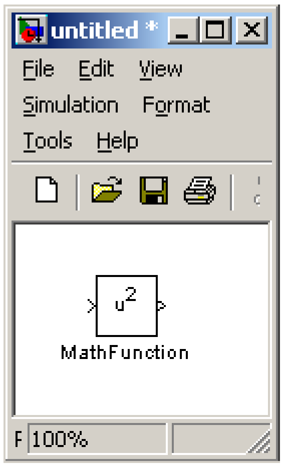
except if the In2 value is equal to zero in which case the output value will be set to zero.

7.3.3.4 Associated EPICS Records

None.

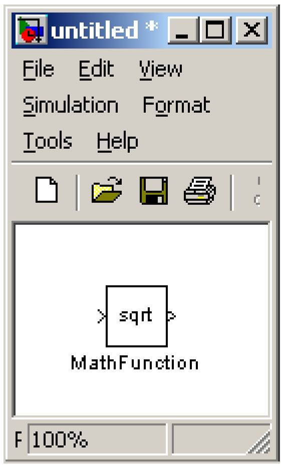
7.3.3.5 Code Examples

The MathFunction module generates the following C code:



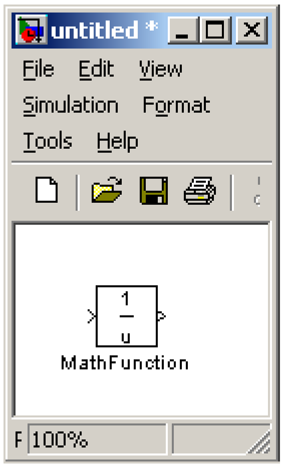
Square:

```
double mathfunction;

// MATH FUNCTION - SQUARE
mathfunction = <In1> * <In1>;

<Out1> = mathfunction;
```



Square root:

```
double mathfunction;

// MATH FUNCTION - SQUARE ROOT
if (<In1> > 0.0) {
        mathfunction = lsqrt(<In1>);
}
else {
        mathfunction = 0.0;
}

<Out1> = mathfunction;
```

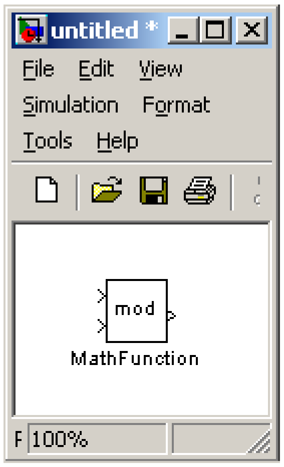


Reciprocal:

```
double mathfunction;

// MATH FUNCTION - RECIPROCAL
if (<In1> != 0.0) {
        mathfunction = 1.0/<In1>;
}
else {
        mathfunction = 0.0;
}

<Out1> = mathfunction;
```

Modulo:

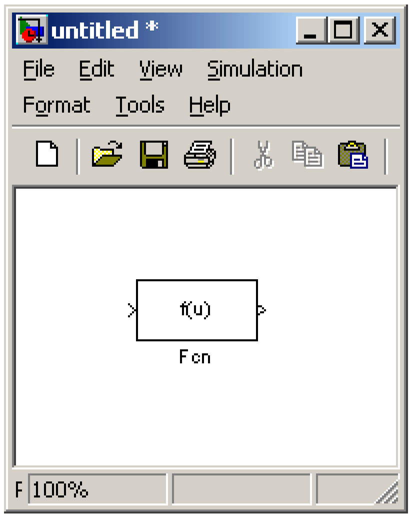
```
double mathfunction;

// MATH FUNCTION - MODULO
if ((int) <In2> != 0) {
        mathfunction = (double) ((int) <In1>%(int) <In2>);
}
else {
        mathfunction = 0.0;
}


<Out1> = mathfunction;
```



7.3.4 In-line (math) function

7.3.4.1 Function

This module is used to include a user defined in-line (math) function in a model.



7.3.4.2 Usage

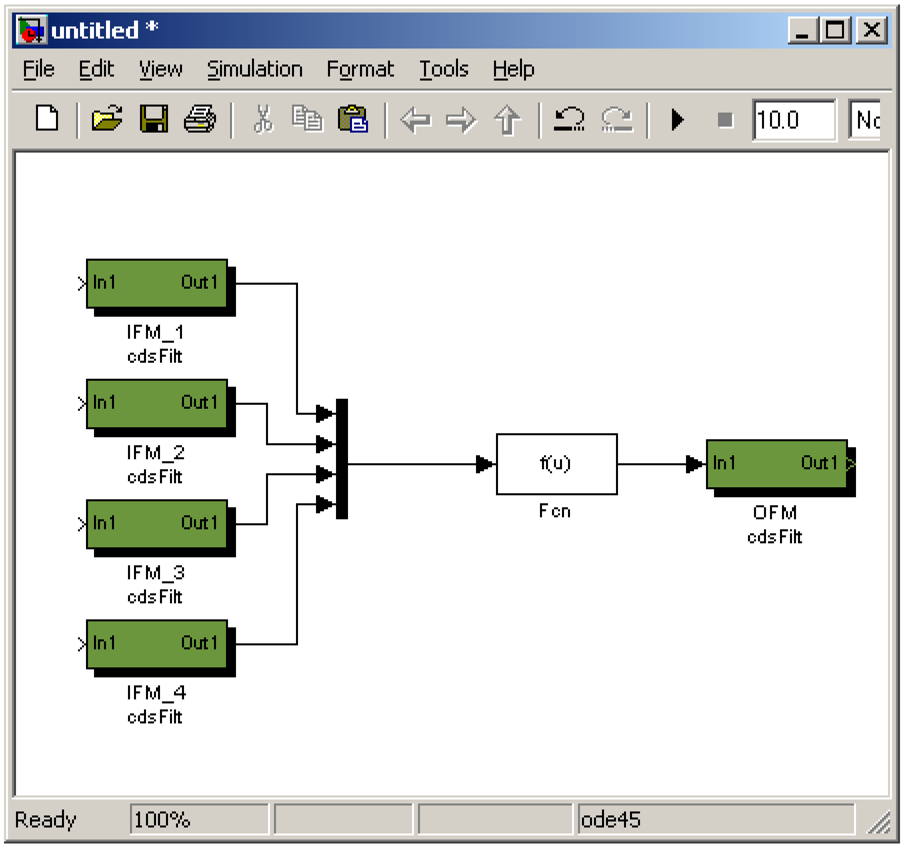
The module supports a number of different types of mathematical functions:

- Polynomials.
- Non-polynomial combinations of variables and constants.
- Sines and cosines.
- Floating-point absolute values.
- log10.
- Square root.
- Combinations of the above.

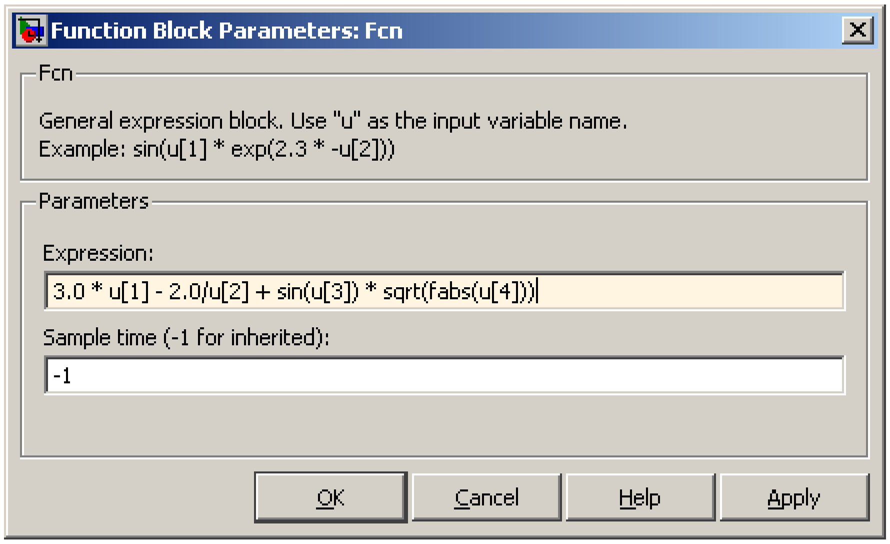
7.3.4.3 Operation

When using this module, place it in the model window and connect the desired number of input variables via a Mux and one output that will pass on the resulting value from the (user defined) function. Double click on the Fcn icon and enter the desired function in the Expression field. The first (top) input variable to the Mux is defined as ‘u[1]’, the second input variable (from the top) is defined as ‘u[2]’, etc. (please note the square brackets). The user defined function can consist of any combination of terms made up of constants multiplied with variables, sine and/or cosine functions, floating-point absolute values, log10 values, and/or square roots.

A (ficticious) example could be as follows (see next page):



Once the function has been defined, click on OK and the function will be incorporated into the model. Please note that it is up to the user to ensure the validity of entered functions and values, e.g., only positive values for logarithms, no negative values for square roots, no divisions by zero, etc. Also, sine and cosine values should, by default, be given in radians. If angles in degrees are desired, replace 'sin' with 'sindeg' and 'cos' with 'cosdeg'.


Function Block Parameters: Fcn

Fcn

General expression block. Use "u" as the input variable name.
Example: sin(u[1] * exp(2.3 * -u[2]))

Parameters

Expression:
3.0 * u[1] - 2.0/u[2] + sin(u[3]) * sqrt(fabs(u[4]))

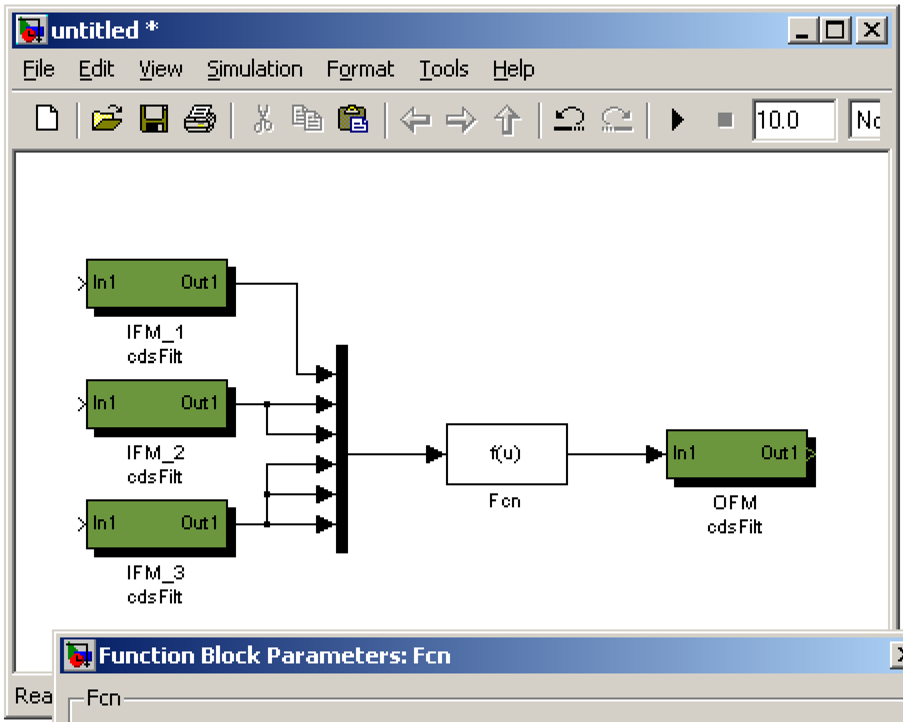
Sample time (-1 for inherited):
-1

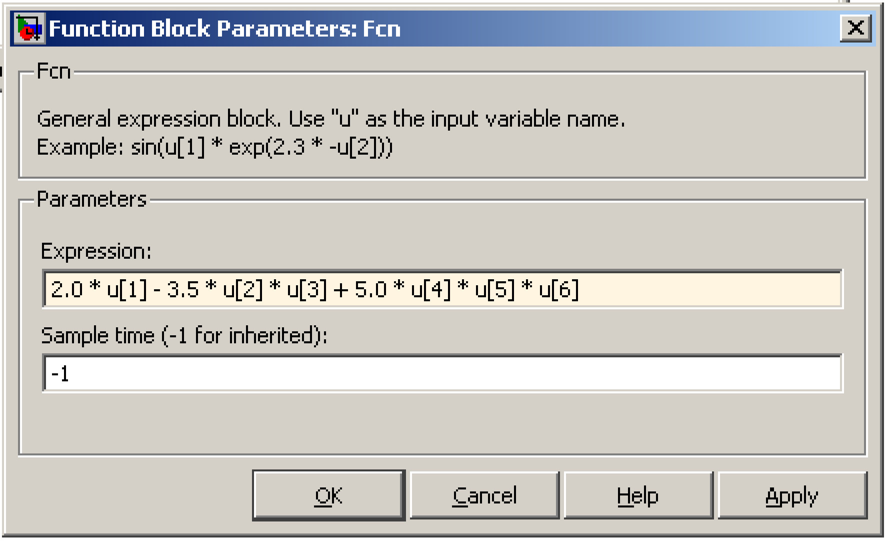
In order to include polynomials, a special technique must be used. This is best explained with an example. Let's assume the following polynomial should be used:

Out = 2.0 * In1 – 3.5 * In2 ** 2 + 5.0 * In3 ** 3

This would require a Mux with six inputs:

In other words, the first input variable ('In1') is connected to the first input to the Mux ('u[1]'), the second input variable ('In2') is connected to the second and third inputs to the Mux (and will be referred to as 'u[2]' and 'u[3]' in the function expression), and the third input variable ('In3') is connected to the fourth, fifth, and sixth inputs to the Mux (referred to as 'u[4]', 'u[5]', and 'u[6]', respectively).




Function Block Parameters: Fcn

Fcn

General expression block. Use "u" as the input variable name.
Example: sin(u[1] * exp(2.3 * -u[2]))

Parameters

Expression:
2.0 * u[1] - 3.5 * u[2] * u[3] + 5.0 * u[4] * u[5] * u[6]

Sample time (-1 for inherited):
-1

7.3.4.4 Associated EPICS Records

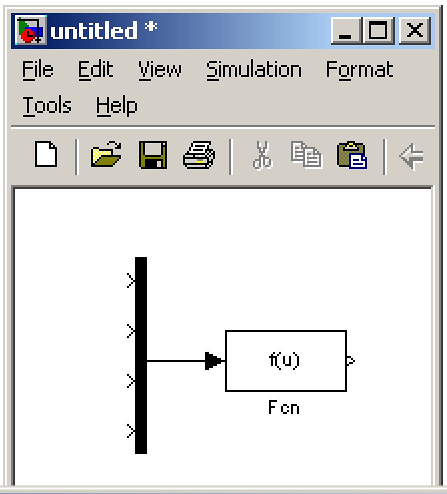
None.

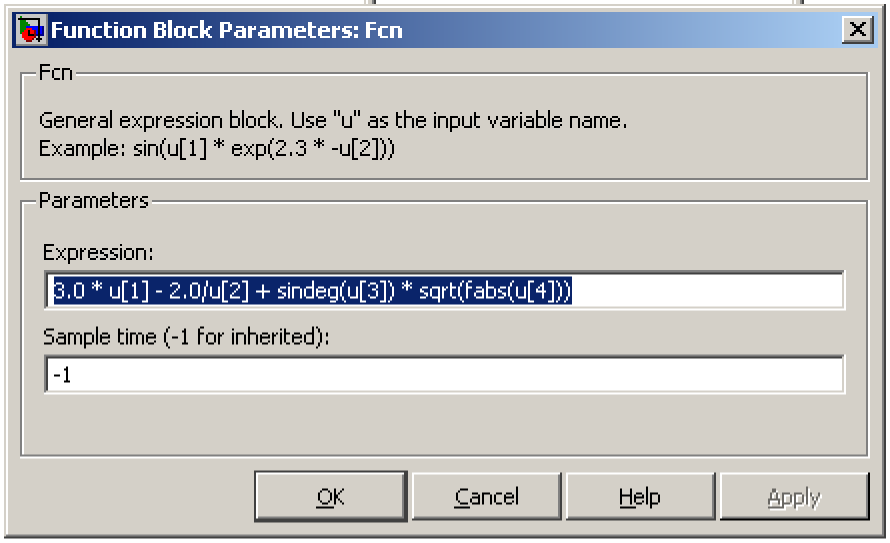
7.3.4.5 Code Examples

The in-line (math) function generates the following C code:

(This first example is identical to the first example in section 7.3.4.3.)

```
double fcn;
double conv = 3.141592654/180.0;
double lcos1, lsin1;

double mux[4];

// MUX
mux[0]= <In1[0]>;
mux[1]= <In1[1]>;
mux[2]= <In1[2]>;
mux[3]= <In1[3]>;

// Inline Function
mux[2] *= conv;
sincos(mux[2], &lsin1, &lcos1);
fcn = 3.0 * mux[0] - 2.0/mux[1] + lsin1 * lsqrt(lfabs(mux[3]));

<Out1> = fcn;
```




Function Block Parameters: Fcn

Fcn

General expression block. Use "u" as the input variable name.
Example: sin(u[1] * exp(2.3 * -u[2]))

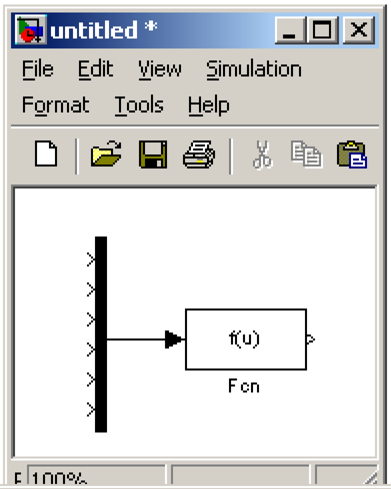
Parameters

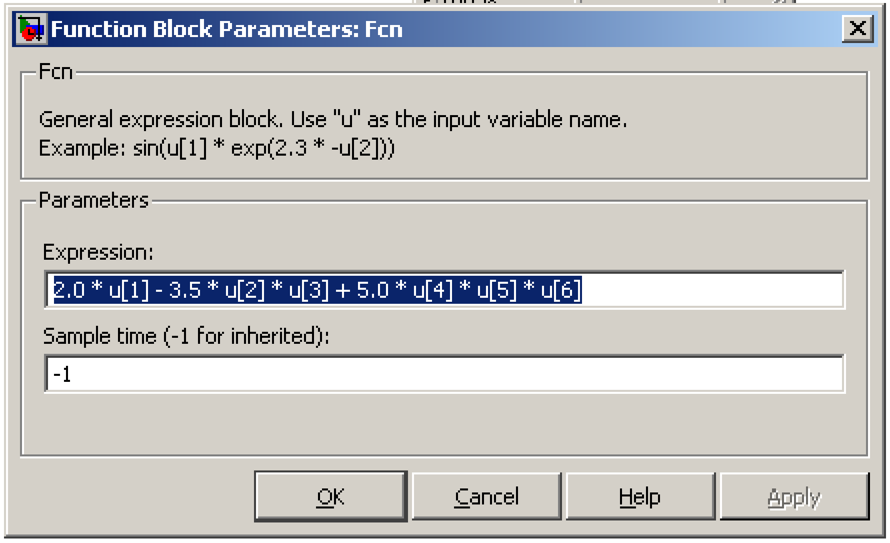
Expression:
3.0 * u[1] - 2.0/u[2] + sindeg(u[3]) * sqrt(fabs(u[4]))

Sample time (-1 for inherited):
-1

(This example is identical to the second example in section 7.3.4.3.)

```
double fcn;

double mux[6];

// MUX
mux[0]= <In1[0]>;
mux[1]= <In1[1]>;
mux[2]= <In1[2]>;
mux[3]= <In1[3]>;
mux[4]= <In1[4]>;
mux[5]= <In1[5]>;

// Inline Function
fcn = 2.0 * mux[0] - 3.5 * mux[1] * mux[2] + 5.0 * mux[3] * mux[4] *
mux[5];

<Out1> = fcn;
```




Function Block Parameters: Fcn

Fcn

General expression block. Use "u" as the input variable name.
Example: sin(u[1] * exp(2.3 * -u[2]))

Parameters

Expression:

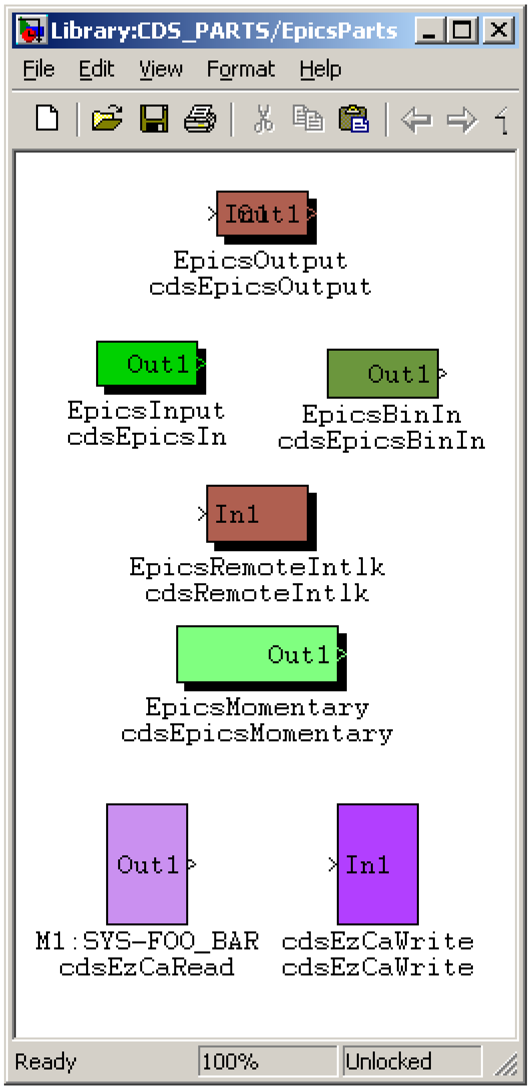
2.0 * u[1] - 3.5 * u[2] * u[3] + 5.0 * u[4] * u[5] * u[6]

Sample time (-1 for inherited):

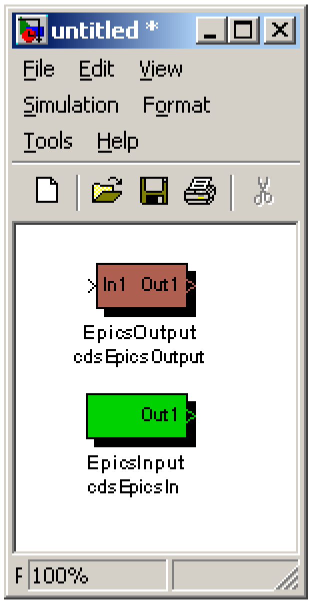
-1

7.4 EPICS Parts

EPICS parts are used to input/output signals from/to the real-time application and EPICS. Some are used primarily to communicate with operator displays, while others are intended to allow multiple FE computers to communicate with each other using EPICS Channel Access (CA) via Ethernet connections.



7.4.1 cdsEpicsOutput/cdsEpicsIn



7.4.1.1 Function

The cdsEpicsOutput module is used to write data into an EPICS channel and the cdsEpicsIn module reads in data from an EPICS channel. *NOTE: The resulting EPICS channels are built on and communicate with EPICS on the local computer. To access EPICS channels on other computers, use the cdsEzCaRead/Write modules.*

7.4.1.2 Usage

For the EpicsOutput, connect the signal to be sent to EPICS via the 'In1' connection. The 'Out1' connection may be used to continue the signal into another RCG part.
For EpicsInput, use the 'Out1' connection to pick up the EPICS data.
For both, modify the name to the desired EPICS channel name.

7.4.1.3 Operation

The RCG will produce local EPICS records with the assigned names and the real-time software will communicate data to/from the EPICS records via shared memory.

7.4.1.4 Associated EPICS Records

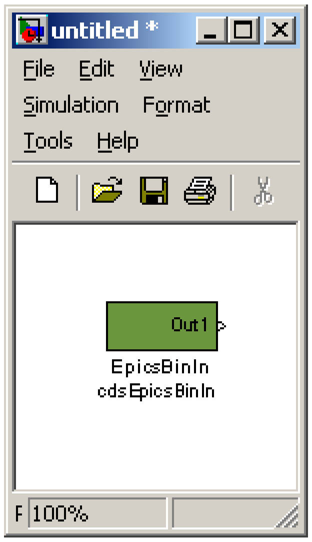
A single 'ai' EPICS record will be produced using the assigned name.

7.4.1.5 Code Examples

The cdsEpicsIn and cdsEpicsOutput modules generate the following C code:

```
void feCode(int cycle, double dWord[ ][32],      /* ADC inputs */
            double dacOut[ ][16],          /* DAC outputs */
            FILT_MOD *dsp_ptr,             /* Filter Mod variables */
            COEF *dspCoeff,                /* Filter Mod coeffs */
            CDS_EPICS *pLocalEpics,  /* EPICS variables */
            int feInit)              /* Initialization flag */
{


<Out1> = pLocalEpics-><Sys>.EpicsInput;


// EpicsOut
pLocalEpics-><Sys>.EpicsOutput = <In1>;
```

7.4.2 cdsEpicsBinIn



7.4.2.1 Function

This part is used to interface a standard EPICS binary input record into the real-time application.

7.4.2.2 Usage

Connect the output to where in EPICS value is to be passed.

7.4.2.3 Operation

Out1 = EPICS value placed in shared memory.

7.4.2.4 Associated EPICS Records

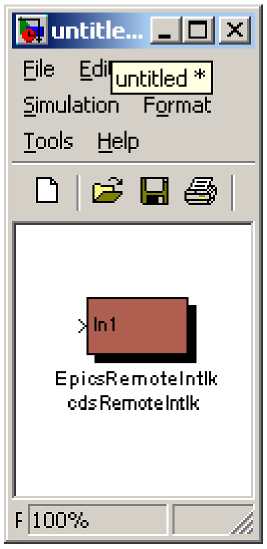
A single 'bi' EPICS record will be produced using the assigned name.

7.4.2.5 Code Example

The cdsEpicsBinIn module generates the following C code:

```
void feCode(int cycle, double dWord[ ][32],      /* ADC inputs */
            double dacOut[ ][16],           /* DAC outputs */
            FILT_MOD *dsp_ptr,              /* Filter Mod variables */
            COEF *dspCoeff,                 /* Filter Mod coeffs */
            CDS_EPICS *pLocalEpics,  /* EPICS variables */
            int feInit)             /* Initialization flag */
{

<Out1>= pLocalEpics-><Sys>.EpicsBinIn;
```

7.4.3 cdsRemoteIntlk



7.4.3.1 Function

7.4.3.2 Usage

7.4.3.3 Operation

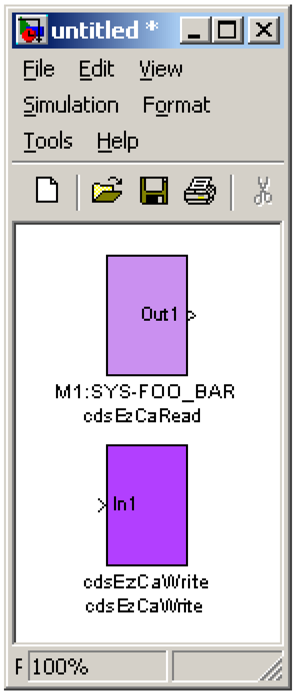
7.4.3.4 Associated EPICS Records

A single ‘ai’ EPICS record will be produced using the assigned name.

7.4.3.5 Code Example

```
// RemoteIntlk
pLocalEpics-><Sys>.EpicsRemoteIntlk = <In1>;
```

7.4.4 cdsEzCaRead/cdsEzCaWrite



7.4.4.1 Function

These blocks are used to communicate data, via EPICS channel access, between real-time code running on separate computers.

7.4.4.2 Usage

Insert the block into the model and modify the name to be the exact name of the remote EPICS channel to be accessed. This must be the full name, in LIGO standard format, including IFO:SYS-.

7.4.4.3 Operation

The EPICS sequencer which supports the real-time code will have EzCaRead/EzCaWrite commands added to obtain/set the desired values via the Ethernet. Values are passed out of/into the real-time code via shared memory.

7.4.4.4 Associated EPICS Records

Each of these two modules will produce a double precision floating-point EPICS channel access record.

7.4.4.5 Code Examples

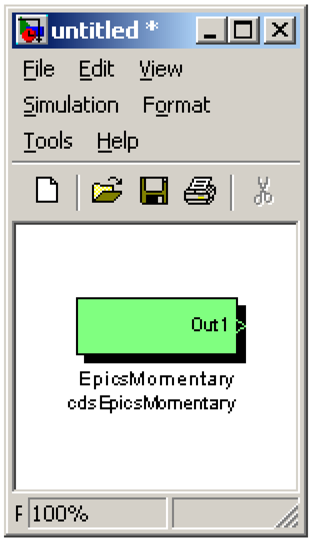
The cdsEzCaRead module generates the following C code:

```
<Out1> = pLocalEpics-><Sys>.<Remote_IFO>_<Remote_Sys>_<Remote_Channel>;
```

The cdsEzCaWrite module generates the following C code:

```
// EzCaWrite
pLocalEpics-><Sys>.<Remote_IFO>_<Remote_Sys>_<Remote_Channel> = <In1>;
```

7.4.5 cdsEpicsMomentary



7.4.5.1 Function

The cdsEpicsMomentary module is used to flip one bit…

7.4.5.2 Usage

…

7.4.5.3 Operation

…

7.4.5.4 Associated EPICS Records

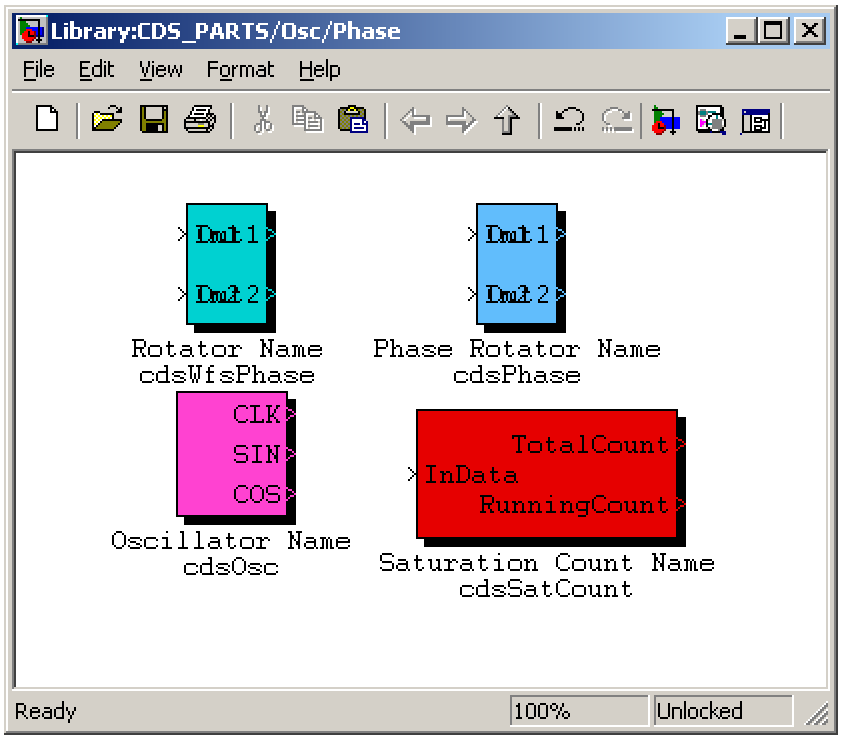
A momentary ‘ai’ EPICS record switch will be produced using the name assigned to this block.

7.4.5.5 Code Example

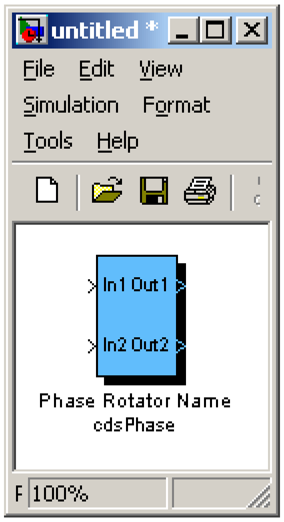
```
static unsigned int epicsmomentary;

if(feInit)
{
      :
epicsmomentary = 0;
      :
} else {

// EpicsMomentary
if (pLocalEpics-><Sys>.EPICSMOMENTARY != 0) {
      epicsmomentary = epicsmomentary ^ pLocalEpics-><Sys>.EPICSMOMENTARY;
      pLocalEpics-><Sys>.EPICSMOMENTARY = 0;
};

<Out1> = epicsmomentary;
```

7.5 Osc/Phase

The Osc/Phase section groups together two different phase rotators, a software oscillator, and a saturation count module.



7.5.1 cdsPhase



7.5.1.1 Function

This block replicates an I&Q phase rotator used in the LIGO LSC control software.

7.5.1.2 Usage

This module is used to change the phase of the input values by a specific phase angle.

7.5.1.3 Operation

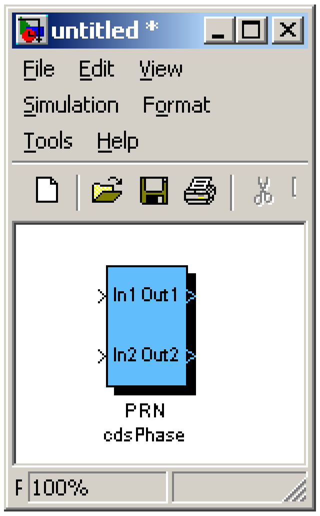
The EPICS code reads in the user variable and calculates the sine and cosine for this entered value. These two values (sinPhase, cosPhase) are then passed to the real-time software, which performs the following calculations:

Out1 = In1 * cosPhase + In2 * sinPhase
Out2 = In2 * cosPhase – In1 * sinPhase

7.5.1.4 Associated EPICS Records

A single ‘ai’ EPICS record is produced to support this module. Entries in this record are in units of degrees.

7.5.1.5 Code Example



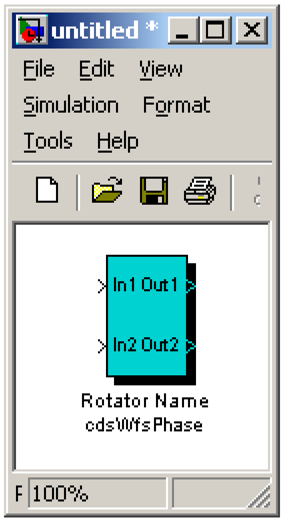
The cdsPhase module generates the following C code:

```
static double prn[2];

// PHASE
prn[0] = (<In1> * pLocalEpics-><Sys>.PRN[1]) +
         (<In2> * pLocalEpics-><Sys>.PRN[0]);
prn[1] = (<In2> * pLocalEpics-><Sys>.PRN[1]) –
         (<In1> * pLocalEpics-><Sys>.PRN[0]);

<Out1> = prn[0];
<Out2> = prn[1];
```

where

pLocalEpics-><Sys>.PRN[0] = sin(pLocalEpics-><Sys>.PRN)
pLocalEpics-><Sys>.PRN[1] = cos(pLocalEpics-><Sys>.PRN)

7.5.2 cdsWfsPhase



7.5.2.1 Function

7.5.2.2 Usage

7.5.2.3 Operation

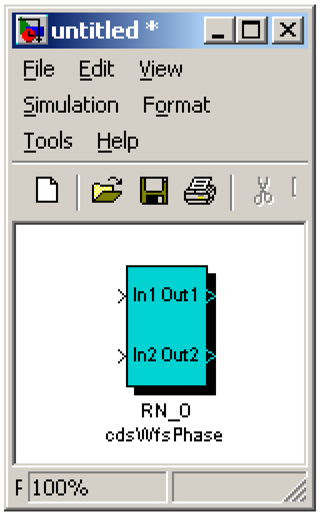
7.5.2.4 Associated EPICS Records

A single 'ai' EPICS record is produced to support this module. Entries in this record are in units of degrees.

7.5.2.5 Code Example

The cdsWfsPhase module generates the following C code:

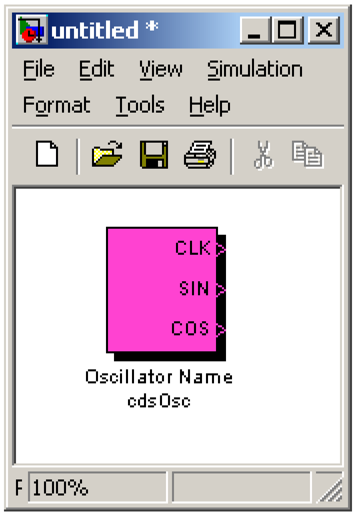
```
static double rn_0[2];

// WFS PHASE
rn_0[0] = (<In1> * pLocalEpics-><Sys>.RN_0[0][0]) –
          (<In2> * pLocalEpics-><Sys>.RN_0[1][0]);
rn_0[1] = (<In2> * pLocalEpics-><Sys>.RN_0[1][1]) –
          (<In1> * pLocalEpics-><Sys>.RN_0[0][1]);

<Out1> = rn_0[0];
<Out2> = rn_0[1];
```



where

```
pLocalEpics-><Sys>.RN_0[0][0] = sin(pLocalEpics-><Sys>.RN_0_r +
  pLocalEpics-><Sys>.RN_0_d)/sin(pLocalEpics-><Sys>.RN_0_d)
pLocalEpics-><Sys>.RN_0[0][1] = cos(pLocalEpics-><Sys>.RN_0_r +
  pLocalEpics-><Sys>.RN_0_d)/sin(pLocalEpics-><Sys>.RN_0_d)
pLocalEpics-><Sys>.RN_0[1][0] = sin(pLocalEpics-><Sys>.RN_0_r)/
  sin(pLocalEpics-><Sys>.RN_0_d)
pLocalEpics-><Sys>.RN_0[1][1] = cos(pLocalEpics-><Sys>.RN_0_r)/
  sin(pLocalEpics-><Sys>.RN_0_d)
```

7.5.3 cdsOsc



7.5.3.1 Function

This block is a software oscillator, developed to support dither locking where two signals with 90 degrees phase rotation are required.

7.5.3.2 Usage

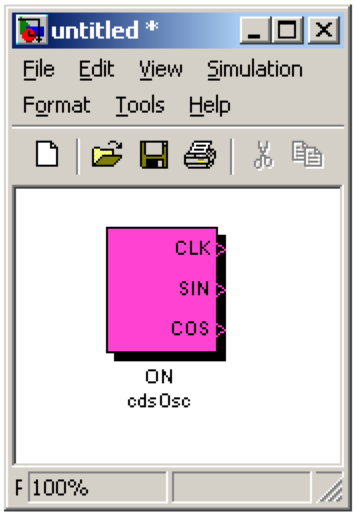
This module is used to produce a sine wave at a specific frequency.

7.5.3.3 Operation

The three outputs are a sine wave at the user requested frequency. The CLK and SIN outputs are in phase with each other and the COS is 90 degrees out of phase. The block internal sine wave varies in amplitude from -1 to 1. The three outputs are then multiplied by their individual gain settings to produce the CLK, SIN and COS outputs.

7.5.3.4 Associated EPICS Records

Four EPICS records are produced for user entries:
_FREQ: Desired frequency in Hz
_CLKGAIN: CLK gain setting
_SINGAIN: SIN gain setting
_COSGAIN: COS gain setting

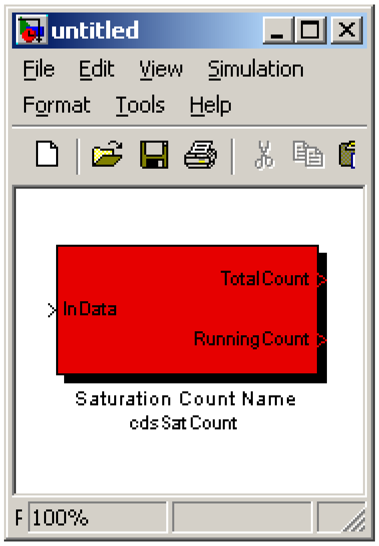


7.5.3.5 Code Example

The cdsOsc module generates the following C code:

```
static double on[3];
static double on_freq;
static double on_delta;
static double on_alpha;
static double on_beta;
static double on_cos_prev;
static double on_sin_prev;
static double on_cos_new;
static double on_sin_new;
double lsinx, lcosx, valx;

if(feInit)
{

on_freq = pLocalEpics-><Sys>.ON_FREQ;
on_delta = 2.0 * 3.14159265358979323846 * on_freq / FE_RATE;
valx = on_delta / 2.0;
sincos(valx, &lsinx, &lcosx);
on_alpha = 2.0 * lsinx * lsinx;
valx = on_delta;
sincos(valx, &lsinx, &lcosx);
on_beta = lsinx;
on_cos_prev = 1.0;
on_sin_prev = 0.0;
} else {

// Osc
on_cos_new = (1.0 - on_alpha) * on_cos_prev - on_beta * on_sin_prev;
on_sin_new = (1.0 - on_alpha) * on_sin_prev + on_beta * on_cos_prev;
on_sin_prev = on_sin_new;
on_cos_prev = on_cos_new;
on[0] = on_sin_new * pLocalEpics-><Sys>.ON_CLKGAIN;
on[1] = on_sin_new * pLocalEpics-><Sys>.ON_SINGAIN;
on[2] = on_cos_new * pLocalEpics-><Sys>.ON_COSGAIN;
if((on_freq != pLocalEpics-><Sys>.ON_FREQ) && ((clock16K + 1) == FE_RATE))
{
        on_freq = pLocalEpics-><Sys>.ON_FREQ;
        on_delta = 2.0 * 3.1415926535897932384626 * on_freq / FE_RATE;
        valx = on_delta / 2.0;
        sincos(valx, &lsinx, &lcosx);
        on_alpha = 2.0 * lsinx * lsinx;
        valx = on_delta;
        sincos(valx, &lsinx, &lcosx);
        on_beta = lsinx;
        on_cos_prev = 1.0;
        on_sin_prev = 0.0;
}

<CLK> = on[0];
<SIN>  = on[1];
<COS> = on[2];

}
```

7.5.4 cdsSatCount



7.5.4.1 Function

The purpose of this block is to count the number of times a channel has saturated since the last time the counter was reset.

7.5.4.2 Usage

This block is used to monitor a data channel in order to keep track of whether or not the input datum is greater than or equal to a saturation threshold value and also keep counts of how often this happens.

7.5.4.3 Operation

Both the TotalCount counter and the RunningCount counter are zeroed on initialization.

The TotalCount counter will keep incrementing (by one per cycle) as long as the absolute value of the channel (input) datum is greater than or equal to the TRIGGER (EPICS input) threshold value. The TotalCount counter can only be reset (to zero) by entering a one in the RESET (EPICS input) switch.

The RunningCount counter will keep incrementing (by one per cycle) as long as the absolute value of the channel (input) datum is greater than or equal to the TRIGGER (EPICS input) threshold value. This counter will be reset (to zero) when the channel (input) datum becomes less than the TRIGGER (EPICS input) threshold value or, conversely, when the TRIGGER (EPICS input) threshold value is modified to a value greater than the channel (input) datum.

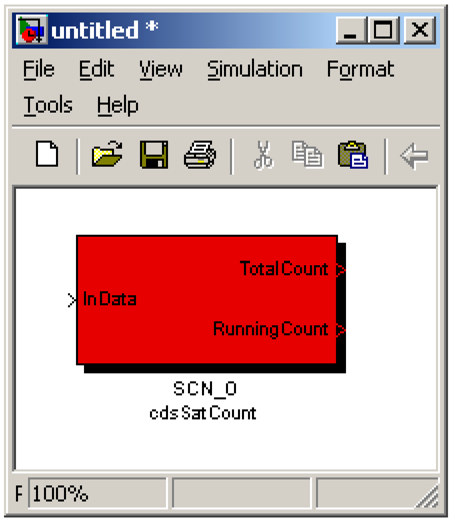
7.5.4.4 Associated EPICS Records

Two EPICS records are produced for user inputs:

_RESET:	This is a momentary RESET switch that zeroes the TotalCount output (when set to one; initial default value is equal to zero and the RESET switch returns to zero after the TotalCount output has been zeroed).
_TRIGGER:	The TotalCount and RunningCount counters (and outputs) will increment as long as the absolute value of the channel (input) datum is greater than or equal to the TRIGGER threshold value (initial default TRIGGER value is equal to zero).

7.5.4.5 Code Example

The cdsSatCount module generates the following C code:

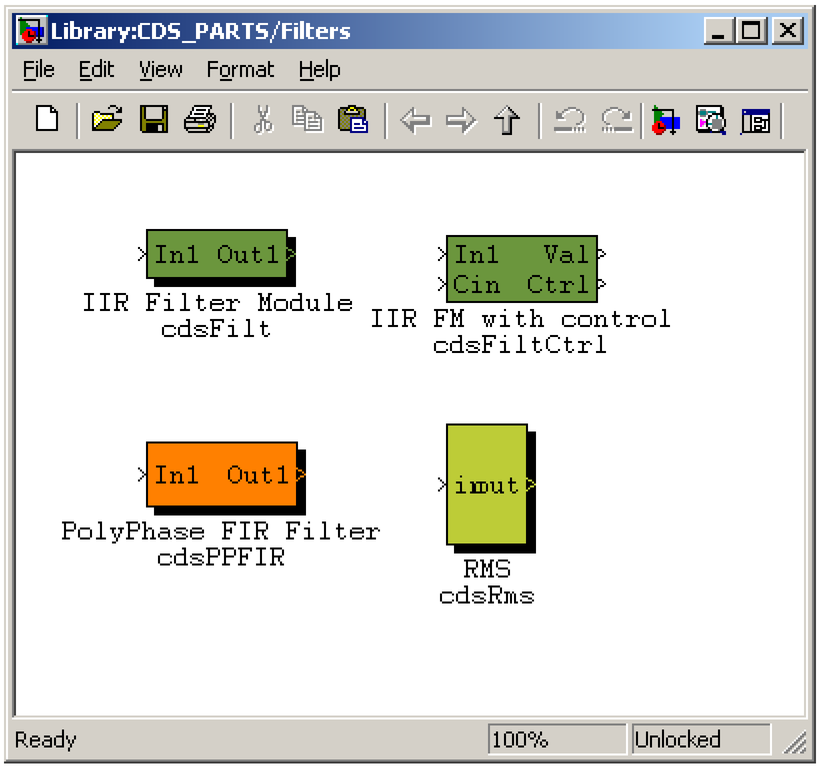


```
int scn_0[2];
static int scn_0_first_time_through = 1;
static int scn_0_total_counter;
static int scn_0_running_counter;

if(feInit)
{

if (scn_0_first_time_through) {
   scn_0_total_counter = 0;
   scn_0_running_counter = 0;
   scn_0_first_time_through = 0;
}

} else {

// SatCount
if (pLocalEpics-><Sys>.SCN_0_RESET == 1) {
   scn_0_total_counter = 0;
   pLocalEpics-><Sys>.SCN_0_RESET = 0;
}
else if (abs(<InData>) >= pLocalEpics-><Sys>.SCN_0_TRIGGER) {
   scn_0_total_counter++;
   scn_0_total_counter%=100000000;
   scn_0_running_counter++;
   scn_0_running_counter%=100000000;
}
else {
   scn_0_running_counter = 0;
}
scn_0[0] = scn_0_total_counter;
scn_0[1] = scn_0_running_counter;

}
<TotalCount> = scn_0[0];
<RunningCount> = scn_0[1];
```

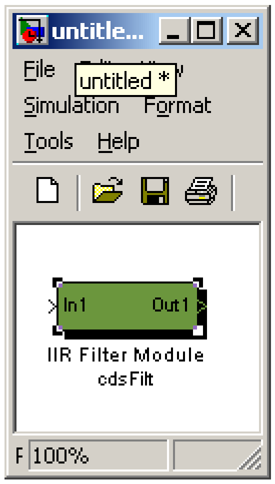
7.6 Filters

The key servo control functions provided by the RCG are in the form of digital filters, as shown in the Filter Parts section.

For most applications, the IIR Filter Module is used. The PolyPhase FIR Filter is designed only for the Ligo HEPI (Hydraulic External Pre-Isolator) controls application and is not intended for general use.



7.6.1 CDS Standard IIR Filter Module



7.6.1.1 Function

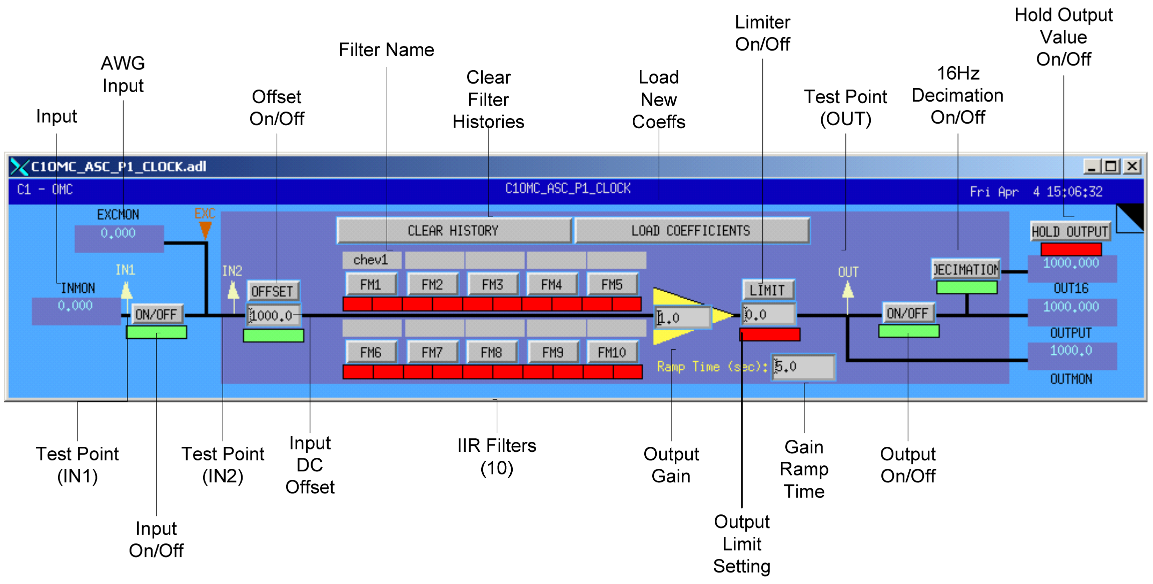
All CDS FE processors use digital Infinite Impulse Response (IIR) filters to perform a majority of their signal conditioning and control algorithm tasks. In order to facilitate their incorporation into FE software and to provide a standard set of DAQ and diagnostic capabilities, the Standard Filter Module (SFM) was developed.

7.6.1.2 Usage

Desired input signal is connected at ‘In1’ and output at ‘Out1’. ‘IIR Filter Module’ name tag is replaced with user name.

7.6.1.3 Operation

To help illustrate the operation of the LIGO CDS Standard Filter Module (SFM), an operator MEDM screen shot is shown below. Signal flow is from Input (left) to Output (right).

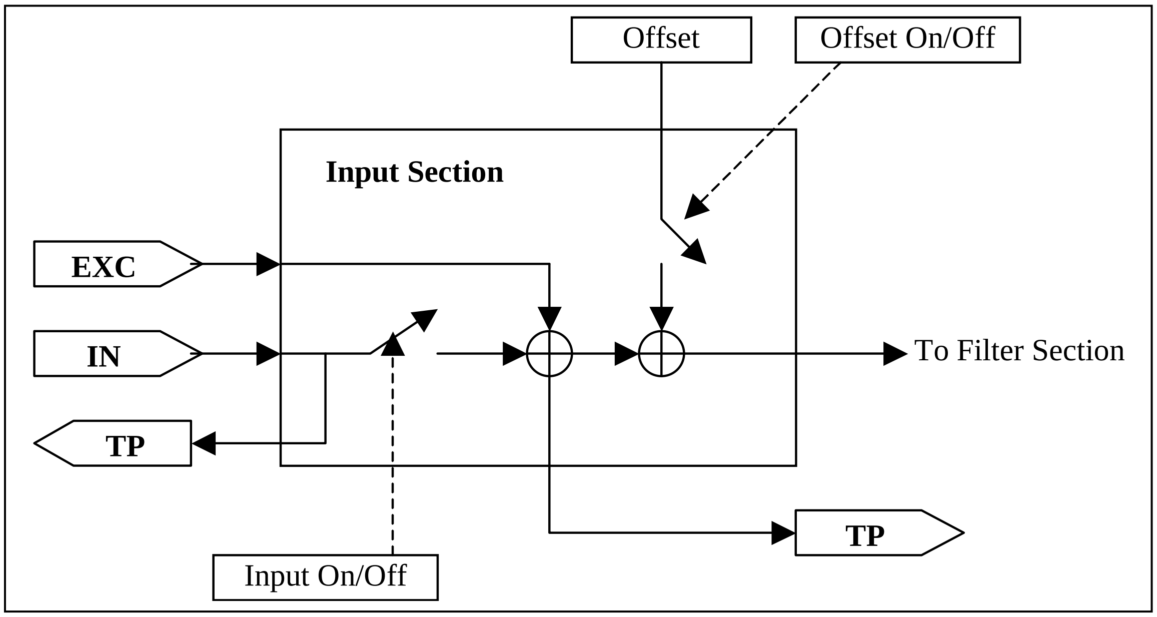


7.6.1.3.1 Input Section

The SFM input is as defined by the user in the MATLAB Simulink model. At run-time, this signal is available to EPICS (_INMON) and is available to diagnostic tools as a test point (_IN1) at the sampling rate of the software. This signal may continue on or be set to zero at this point by use of the Input On/Off switch.

Each SFM also has an excitation signal input available from the Arbitrary Waveform Generator (AWG). This signal is available for EPICS (_EXCMON). The AWG signal is summed with the input signal, and available to diagnostic tools as a second test point (_IN2).

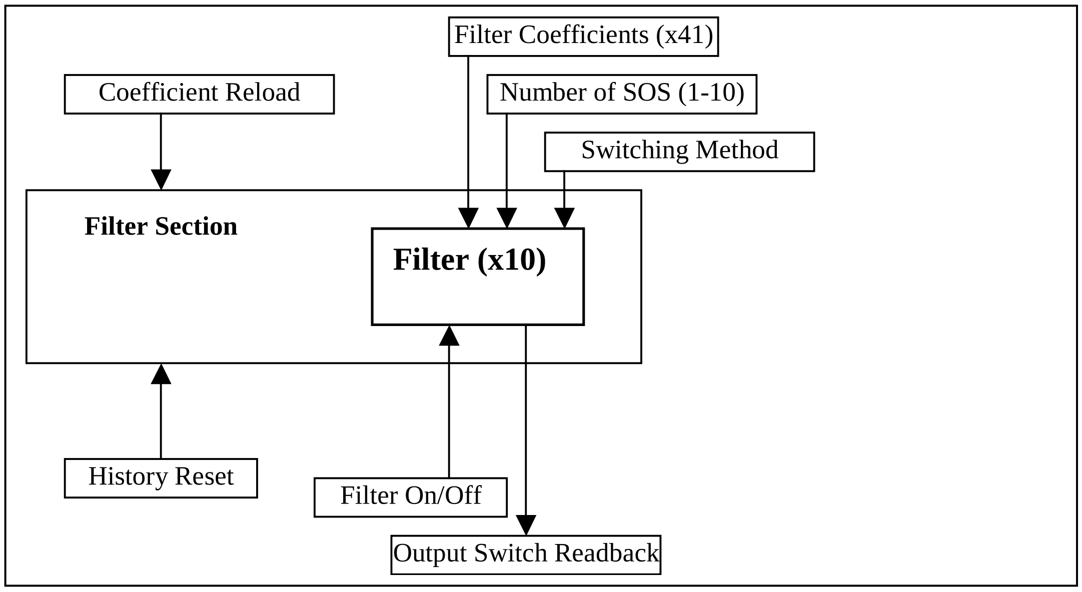
To this resulting signal, a DC offset may be added (Input DC Offset) and this offset may be turned on/off via the Offset on/off switch. The sum of the input, AWG and offset signal is then fed to the IIR filtering section.



7.6.1.3.2   Filtering Section

The filter section may have up to 10 IIR filters defined, with up to 10 Second Order Sections (SOS) each. The software allows for any/all of these filters to be redefined "on the fly", i.e., an FE process does not need to be rebooted, restarted or otherwise interrupted from its tasks during reconfiguration.

Each filter within an SFM may be individually turned on/off during operation. Various types of input/output switching may be defined for each individual filter.



The filter coefficients and switching properties are defined in a text file produced by the *foton* tool. Filter coefficient files used by the SFM must be located in the */cvs/cds/<site>/chans* directory. This file contains:

- The names of all SFMs defined within an FE processor. Each SFM within a front end is given a unique name in the EPICS sequencer software used to download the SFM coefficients to the front end. These names must be provided in this file for use by *foton*. This is done by listing the SFM names after the keyword 'MODULES'. As an example, from the LSC FE file:
  - # MODULES DARM MICH PRC CARM MICH_CORR
  - # MODULES BS RM AS1_I
- A line (or lines) for each filter within an SFM, describing filter attributes and coefficients. These lines must contain the information listed in the following table, in the exact order given in the table.

Field	Description
SFM Name	The EPICS name of the filter module to which the remaining parameters are to apply.
Filter Number	The number of the filter (0-9) within the given SFM to which the remaining parameters are to apply.
Filter Switching	As previously mentioned, individual filters may have different switching capabilities set. This two digit number describes how the filter is to switch on/off. This number is calculated by input_switch_type x 10 + output_switch_type. The supported values for input switching are: • 0 – Input is always applied to filter. • 1 – Input switch will switch with output switch. When filter output switch goes to 'OFF', all filter history variables will be set to zero. Four types of output switching are supported. These are: • 0 – Immediate. The output will switch on or off as soon as commanded. • 1 – Ramp: The output will ramp up over the number of cycles defined by the RAMP field. • 2 – Input Crossing: The output will switch when the filter input and output are within a given value of each other. This value is contained in the RAMP field. • 3 – Zero Crossing: The output will switch when the filter input crosses zero.
Number of SOS	This field contains the number of Second Order Sections in this filter.
RAMP	The contents of this field are dependent on the Filter Switching type.
Timeout	For type 2 and 3 filter output switching (input and zero crossing), a time-out value must be provided (in FE cycles). If the output switching requirements are not met within this number of cycles, the output will switch anyway.
Filter Name	This name will be printed to the EPICS displays which have that filter. It is basically a comment field.
Filter Gain	Overall gain term of the filter.
Filter Coefficients	The coefficients which describe the filter design.

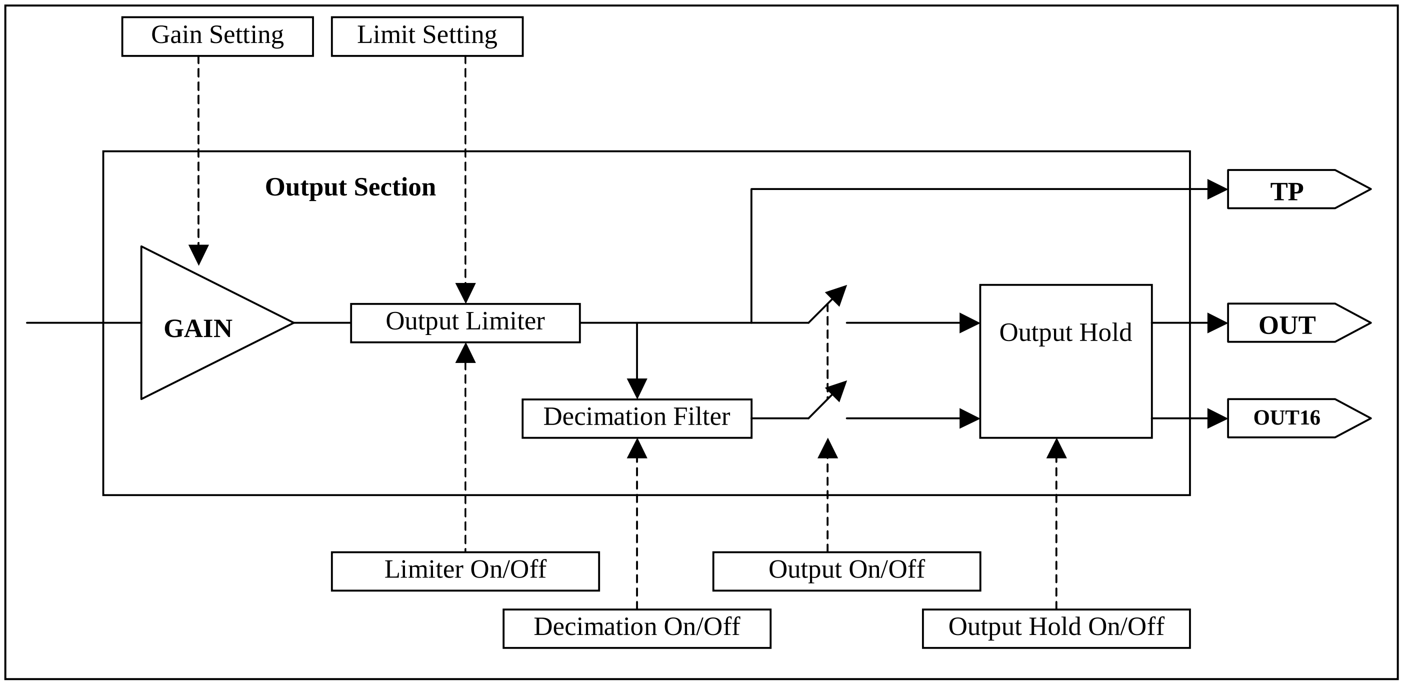
A skeleton coefficient file is produced the first time 'make-install' is invoked after compiling a model file. Thereafter, whenever 'make-install' is executed, the install process will make a back-up of the present coefficient file, then patch the present file with any new filter modules or renaming of filter modules.

7.6.1.3.3 Output Section

The following figure shows the output section. The output section provides for:

- A variable gain to be applied to the filter section output. This gain may be ramped over time from one setting to another by setting the gain ramp time.
- This output to be limited to a selected value (the output limiter can be switched on or off).

- A GDS TP. This TP is always on, regardless of whether the output is turned on or off.
- Ability to turn output on or off.
- A decimation filter to provide a 16Hz output (typically used by EPICS; the decimation filter can be switched on or off).
- A “hold” output feature. When enabled, the output of the SFM will be held to its present value.



7.6.1.4 Associated EPICS Records

For each filter module, the following EPICS records are produced, with the filter name as the prefix:

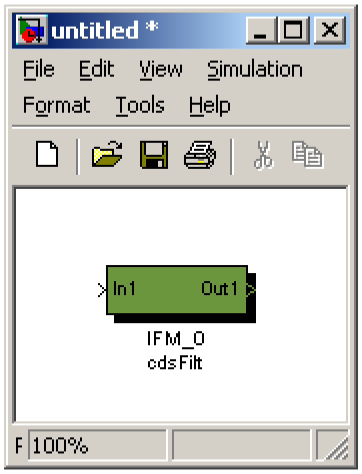
_INMON = Filter module input value (RO)
_EXCMON = Filter module excitation signal input value (RO)
_OFFSET = User settable offset value (W/R)
_GAIN = Filter module output gain (W/R)
_TRAMP = Gain ramping time, in seconds (W/R)
_LIMIT = User defined filter module output limit (W/R)
_OUTMON = Output test-point value (RO)
_OUT16 = Filter module output, decimation filtered to 16Hz (RO)
_OUTPUT = Filter module output value (RO)
_SW1 = Momentary filter switch selections, lower 16 bits (WO)
_SW2 = Momentary filter switch selections, upper 16 bits (WO)
_RSET = Momentary clear filter history switch (WO)
_SW1R = Filter switch read-backs, lower 16 bits (RO)
_SW2R = Filter switch read-backs, upper 16 bits (RO)
_SW1S = Saved filter switch selections, lower 16 bits (RO)
_SW2S = Saved filter switch selections, upper 16 bits (RO)
_Name00 thru _Name09 = Individual filter names, as defined in the coefficient file (RO)

7.6.1.5 Code Example

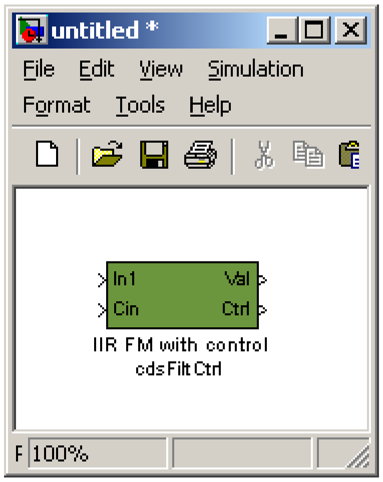
The cdsFilt module generates the following C code:

```
double ifm_0;

// FILTER MODULE
ifm_0 = filterModuleD(dsp_ptr,dspCoeff,IFM_0,<In1>,0);

<Out1> = ifm_0;
```

(The IFM_0 parameter in the filterModuleD function call above is a constant containing a unique filter module id. number.)



7.6.2 IIR Filter Module with Control



7.6.2.1 Function

This module is a standard filter module, with the addition that the SFM switch and filter status are output and a second input has been added.

7.6.2.2 Usage

The additional input must be connected to ground or some other module (e.g., cdsEpicsIn) for the code to compile. The additional control output is used to provide some downstream control or decision making based on the switch settings within the SFM. Typically this output is tied to a bitwise operator to select the desired bits, often to then go to binary output modules to switch relays based on filters being on/off.

7.6.2.3 Operation

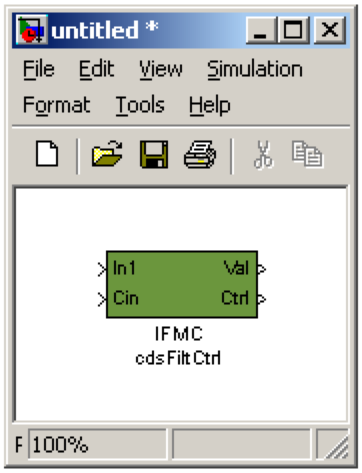
In addition to the SFM operation, this block outputs the internal switch information in the form of a 32-bit integer. The bits of this integer are defined in the following table.

Bit	Name	Description
0	Coeff Reset	This is a momentary bit. When set, the EPICS CPU will read in new SFM coeffs from file and send this information to the FE via the RFM network. The FE SFM will read and load new filter coefficients from RFM.
1	Master Reset	Momentary; when set, SFM will reset all filter history buffers.
2	Input On/Off	Enables/disables signal input to SFM.
3	Offset Switch	Enables/disables application of SFM input offset value.
Even bits 4-22	Filter Request	Set to one when an SFM filter is requested ON, or zero when SFM filter requested OFF (bit 4 is associated with filter module 1, bit 6 with filter module 2, etc.).
Odd bits 5-23	Filter Status	Set to one by SFM when an SFM filter is ON, or zero when SFM filter is OFF (bit 5 is associated with filter module 1, bit 7 with filter module 2, etc.).
24	Limiter Switch	Enables/disables application of SFM output limit value.
25	Decimation Switch	Enables/Disables application of decimation filter to SFM OUT16 calculation.
26	Output Switch	Enables/Disables SFM output (SFM OUT and OUT16 variables)
27	Hold Output	If (!bit 26 && bit27), SFM OUT will be held at last value.
28	Gain Ramp	If set, gain of filter module != requested gain. This bit is set when SFM gain is ramping to a new gain request.

7.6.2.4 Associated EPICS Records

Same as cdsFilt module.

7.6.2.5 Code Example

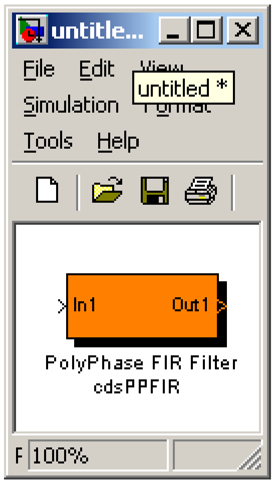


The cdsFiltCtrl module generates the following C code:

```
double ifmc;

// FILTER MODULE
ifmc = filterModuleD(dsp_ptr,dspCoeff,IFMC,<In1>,<Cin>);

<Val> = ifmc;

<Ctrl> = dsp_ptr->inputs[IFMC].opSwitchP|
((0x4|0x8|0x1000000|0x2000000|0x4000000|0x8000000) &
dsp_ptr->inputs[IFMC].opSwitchE);
```

(The IFMC parameter in the filterModuleD function call above is a constant containing a unique filter module id. number.)

7.6.3 PolyPhase FIR Filter



7.6.3.1 Function

This module allows the use of Polyphase FIR (Finite Impulse Response) filters, typically used in seismic isolation system controls.

7.6.3.2 Usage

This part is placed into the model and functions exactly as the cdsFilter part. To load an FIR at runtime, a separate coefficient file must be provided for FIR filters (*/cvs/cds/site/chans/modelName.fir*).
N.B. The sample rate must be either 2K or 4K when PolyPhase FIR Filters are being used.

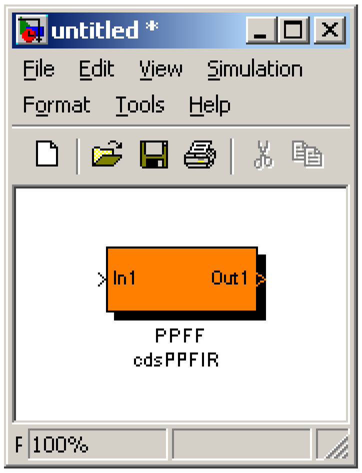
7.6.3.3 Operation

Use of this part simply sets a compiler flag to allow the use of FIR filters. In all other respects, it functions in the same way as the cdsFilter part described previously. In fact, this part allows a mix of IIR and FIR filters to be assigned to the 10 available digital filters within the module. The difference between IIR and FIR is determined by the runtime software by the number of coefficients loaded (>10 SOS = FIR).

7.6.3.4 Associated EPICS Records

Same as cdsFilt module.

7.6.3.5 Code Example

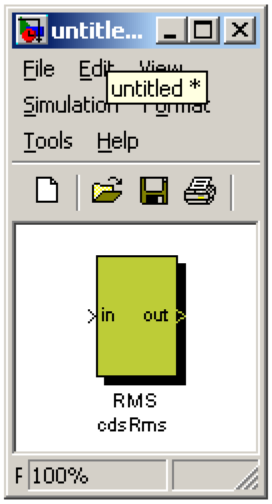


The cdsPPFIR module generates the following C code:

```
double ppff;

// FILTER MODULE
ppff = filterModuleD(dsp_ptr,dspCoeff,PPFF,<In1>,0);

<Out1> = ppff;
```

(The PPFF parameter in the filterModuleD function call above is a constant containing a unique filter module id. number.)

7.6.4 RMS Filter



7.6.4.1 Function

This block computes the RMS value of the input signal.

7.6.4.2 Usage

This module is used to calculate an RMS value.

7.6.4.3 Operation

The output value is the RMS value of the input value, within the limits of ±2000 counts.

7.6.4.4 Associated EPICS Records

None.

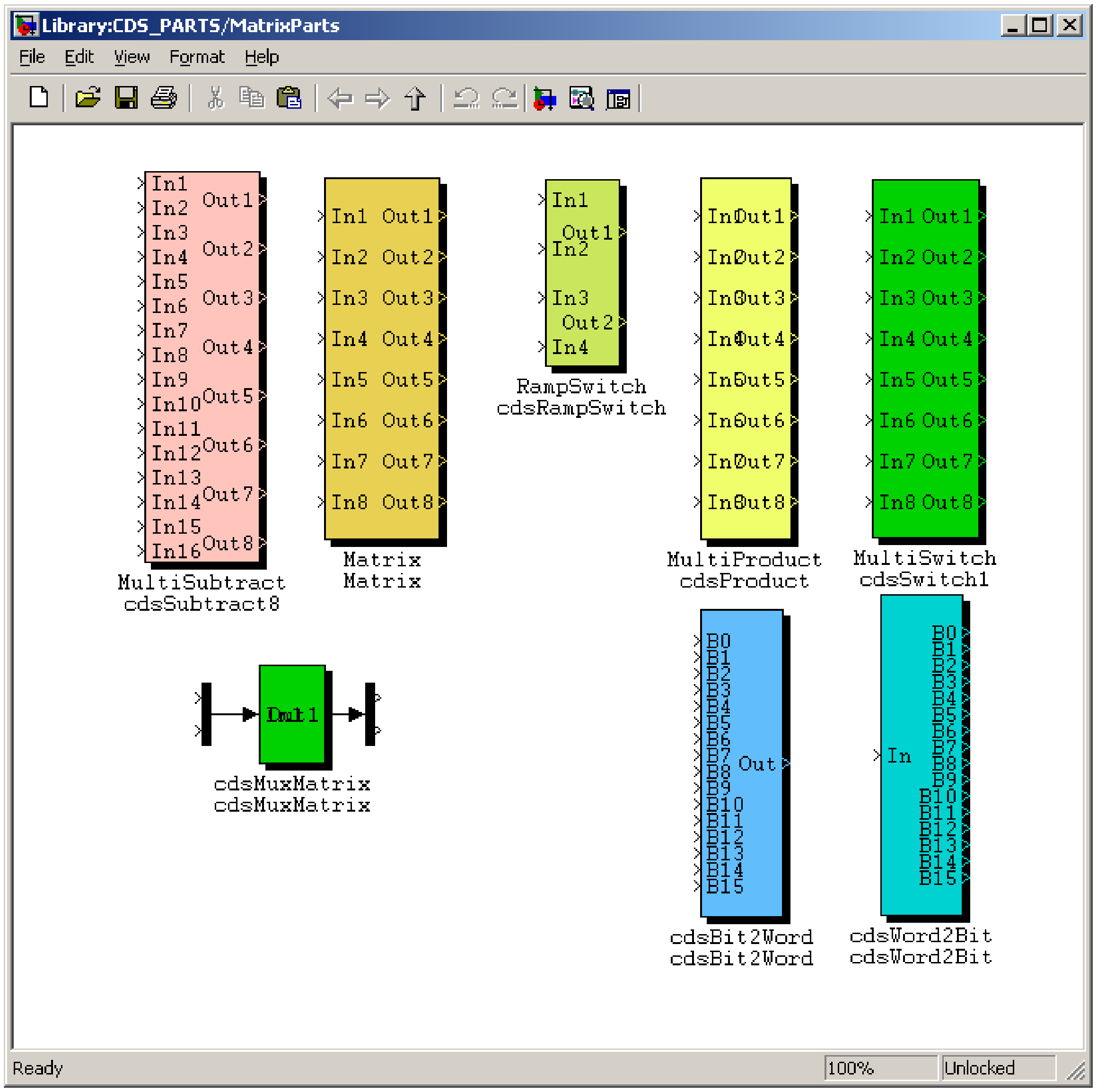
7.6.4.5 Code Example

The cdsRms module generates the following C code:

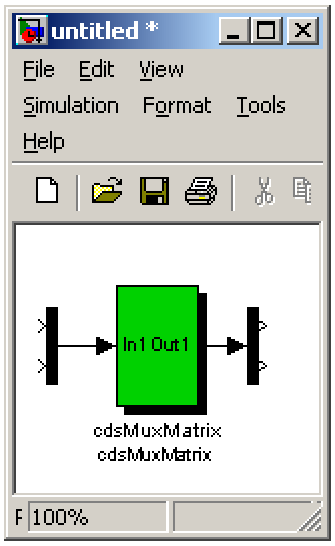
```
float rms;
static float rms_avg;

if(feInit)
{

rms_avg = 0.0;

} else {

// RMS
rms = <in>;
if(rms > 2000) rms = 2000;
if(rms < -2000) rms = -2000;
rms = rms * rms;
rms_avg = rms * .00005 + rms_avg * 0.99995;
rms = lsqrt(rms_avg);

<out> = rms;
}
```

7.7 Matrix Parts

Matrix parts are those which perform calculations based on array data. The most commonly used is the cdsMuxMatrix part.



7.7.1 cdsMuxMatrix



7.7.1.1 Function

The primary function of this block is to produce output signals based on the scaling and addition of various input signals.

7.7.1.2 Usage

Inputs are connected via the Mux part and outputs are connected via the Demux part. The number of connections available at the input/output may be modified to any size by double clicking on the Mux/Demux parts and modifying the number of connection fields in the pop-up window.

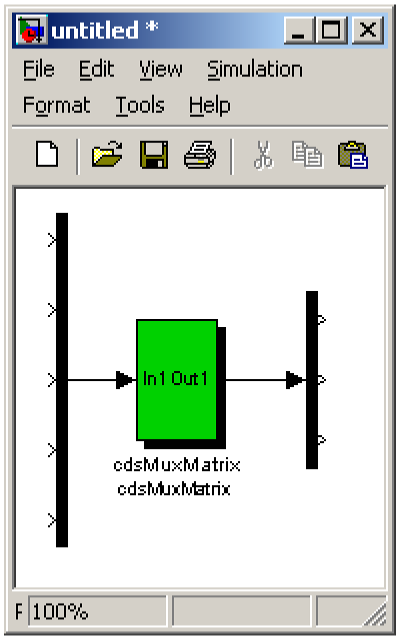
7.7.1.3 Operation

Basic code function is:
Output[1] =
   Input[1] * Matrix_11 + Input[2] * Matrix_12 + Input[n] * Matrix_1n,
where Matrix_xx is an EPICS entry field.

7.7.1.4 Associated EPICS Records

The RCG will produce an A x B matrix of EPICS records for use as input variables, where B is the number of inputs and A is the number of outputs. The EPICS record names will be in the form of PARTNAME_AB, starting at PARTNAME_11.

7.7.1.5 Code Example

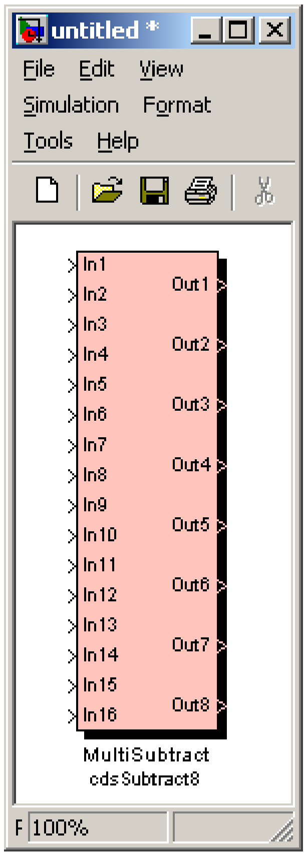


The cdsMuxMatrix module generates the following C code:

```
int ii;

double demux[3];
double mux[5];
double cdsmuxmatrix[3];

// MUX
mux[0]= <In1[0]>;
mux[1]= <In1[1]>;
mux[2]= <In1[2]>;
mux[3]= <In1[3]>;
mux[4]= <In1[4]>;

// MuxMatrix
for(ii=0;ii<3;ii++)
{
cdsmuxmatrix[ii] =
        pLocalEpics-><Sys>.cdsMuxMatrix[ii][0] * mux[0] +
        pLocalEpics-><Sys>.cdsMuxMatrix[ii][1] * mux[1] +
        pLocalEpics-><Sys>.cdsMuxMatrix[ii][2] * mux[2] +
        pLocalEpics-><Sys>.cdsMuxMatrix[ii][3] * mux[3] +
        pLocalEpics-><Sys>.cdsMuxMatrix[ii][4] * mux[4];
}
```

```
// DEMUX
demux[0]= cdsmuxmatrix[0];
demux[1]= cdsmuxmatrix[1];
demux[2]= cdsmuxmatrix[2];

<Out1[0]> = demux[0];
<Out1[1]> = demux[1];
<Out1[2]> = demux[2];
```

7.7.2 MultiSubtract



7.7.2.1 Function

This module is a group of subtractions, packaged into a single part.

7.7.2.2 Usage

Connect all input and output connectors. (N.B. All 16 inputs must be connected to other modules in order for this module to compile.)

7.7.2.3 Operation

This module subtracts pairs of inputs (16) and produces 8 outputs, e.g., Out1 = In2 – In1, Out2 = In4 – In3, etc.

7.7.2.4 Associated EPICS Records

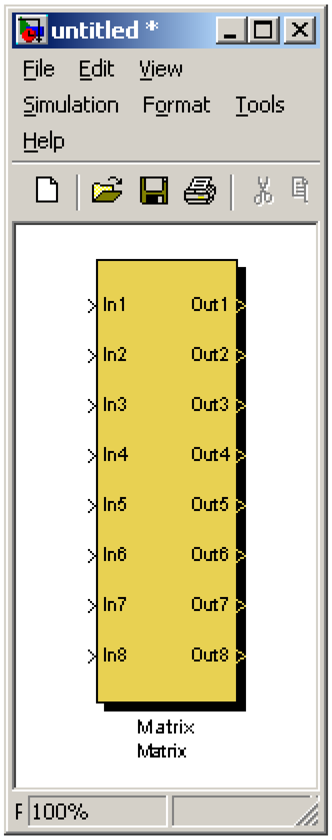
None.

7.7.2.5 Code Example

The cdsSubtract8 module generates the following C code:

```
double multisubtract[16];

// DiffJunc
multisubtract[0] = <In2> - <In1>;
multisubtract[1] = <In4> - <In3>;
multisubtract[2] = <In6> - <In5>;
multisubtract[3] = <In8> - <In7>;
multisubtract[4] = <In10> - <In9>;
multisubtract[5] = <In12> - <In11>;
multisubtract[6] = <In14> - <In13>;
multisubtract[7] = <In16> - <In15>;

<Out1> = multisubtract[0];
<Out2> = multisubtract[1];
<Out3> = multisubtract[2];
<Out4> = multisubtract[3];
<Out5> = multisubtract[4];
<Out6> = multisubtract[5];
<Out7> = multisubtract[6];
<Out8> = multisubtract[7];
```

7.7.3 Matrix



7.7.3.1 Function

The output values produced by this module are made up of the input values multiplied by scale factors and added together.

7.7.3.2 Usage

This module has been replaced by the cdsMuxMatrix module. The Matrix module should NOT be used!

7.7.3.3 Operation

Each input value is multiplied by a scale factor (supplied via EPICS records), after which the resulting values are added together and assigned to the output values.

7.7.3.4 Associated EPICS Records

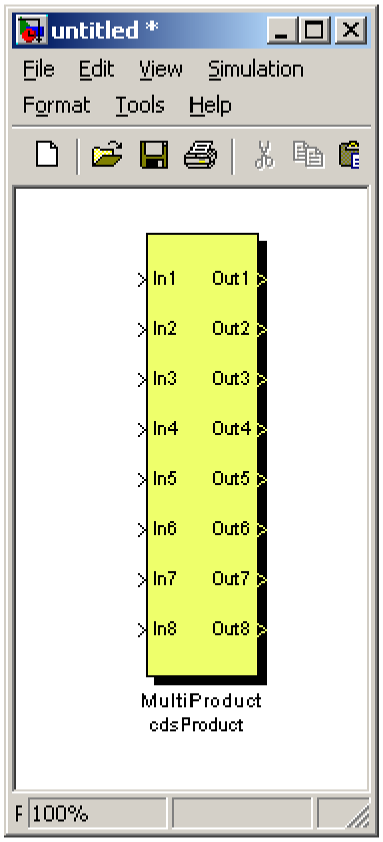
A matrix of A x B EPICS records (where B is the number of inputs and A is the number of outputs) is produced by the Real-Time Code Generator.

7.7.3.5 Code Example

The Matrix module generates the following C code:

```
int ii;

double matrix[8][8];

// Matrix
for(ii=0;ii<8;ii++)
{
matrix[1][ii] =
        pLocalEpics-><Sys>.Matrix[ii][0] * <In1> +
        pLocalEpics-><Sys>.Matrix[ii][1] * <In2> +
        pLocalEpics-><Sys>.Matrix[ii][2] * <In3> +
        pLocalEpics-><Sys>.Matrix[ii][3] * <In4> +
        pLocalEpics-><Sys>.Matrix[ii][4] * <In5> +
        pLocalEpics-><Sys>.Matrix[ii][5] * <In6> +
        pLocalEpics-><Sys>.Matrix[ii][6] * <In7> +
        pLocalEpics-><Sys>.Matrix[ii][7] * <In8>;
}

<Out1> = matrix[1][0];
<Out2> = matrix[1][1];
<Out3> = matrix[1][2];
<Out4> = matrix[1][3];
<Out5> = matrix[1][4];
<Out6> = matrix[1][5];
<Out7> = matrix[1][6];
<Out8> = matrix[1][7];
```

7.7.4 MultiProduct



7.7.4.1 Function

The purpose of this block is to multiply up to eight inputs by a single input gain setting. Whenever a gain setting is changed, this block will ramp the gain from the present to a new setting over the user defined time interval.

7.7.4.2 Usage

The 8 inputs and outputs are connected, either to other signals or terminators. The gain multiplier comes from EPICS.

7.7.4.3 Operation

The code for this block will multiply all inputs by the gain setting and produce the results at the corresponding outputs. If the gain is changed, the code will ramp the gain value over the requested ramp time.

7.7.4.4 Associated EPICS Records

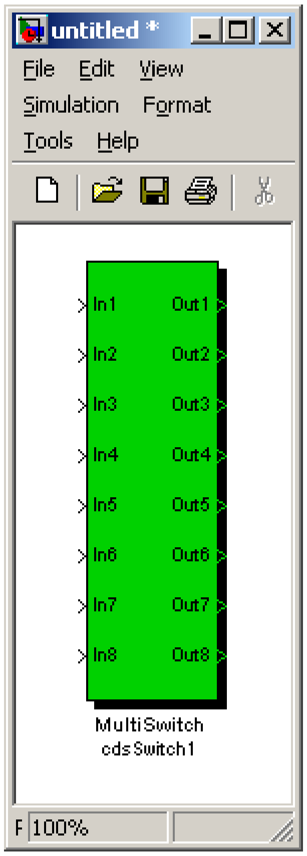
<block name>: Gain to be applied to all channels.
_TRAMP: Time (seconds) over which to ramp any gain changes.
_RMON: Return status code from gainRamp function (not used).

7.7.4.5 Code Example

The cdsProduct module generates the following C code:

```
double multiproduct[8];
float MultiProduct_CALC;

// PRODUCT
pLocalEpics-><Sys>.MultiProduct_RMON = gainRamp(pLocalEpics-><Sys>.MultiProduct,
                          pLocalEpics><Sys>.MultiProduct_TRAMP,0,&MultiProduct_CALC);

multiproduct[0] = MultiProduct_CALC * <In1>;
multiproduct[1] = MultiProduct_CALC * <In2>;
multiproduct[2] = MultiProduct_CALC * <In3>;
multiproduct[3] = MultiProduct_CALC * <In4>;
multiproduct[4] = MultiProduct_CALC * <In5>;
multiproduct[5] = MultiProduct_CALC * <In6>;
multiproduct[6] = MultiProduct_CALC * <In7>;
multiproduct[7] = MultiProduct_CALC * <In8>;

<Out1> = multiproduct[0];
<Out2> = multiproduct[1];
<Out3> = multiproduct[2];
<Out4> = multiproduct[3];
<Out5> = multiproduct[4];
<Out6> = multiproduct[5];
<Out7> = multiproduct[6];
<Out8> = multiproduct[7];
```

7.7.5 MultiSwitch



7.7.5.1 Function

This block allows simultaneous on/off switching of up to 8 signals via a single EPICS input record.

7.7.5.2 Usage

This module is used to connect up to eight inputs that are either passed through to the outputs (if the associated EPICS record is set to one) or switched off (if the EPICS record is set to zero).

7.7.5.3 Operation

When the associated EPICS record is set to ‘1’, ‘In1’ thru ‘In8’ are passed straight through to ‘Out1’ thru ‘Out8’. If the EPICS record is set to zero, ‘Out1’ through ‘Out8’ become zero.

7.7.5.4 Associated EPICS Records

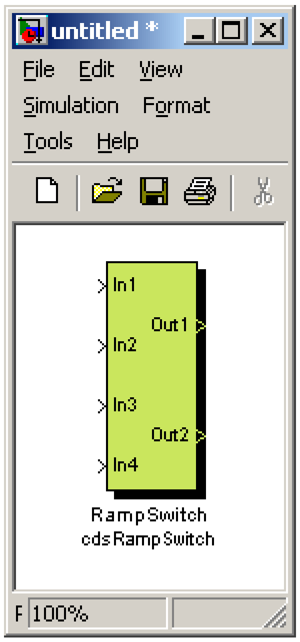
The RCG produces a single EPICS ‘bi’ record with the name given to this part by the user.

7.7.5.5 Code Example

The cdsSwitch1 module generates the following C code:

```
int ii;

double multiswitch[8];

// MultiSwitch
multiswitch[0] = <In1>;
multiswitch[1] = <In2>;
multiswitch[2] = <In3>;
multiswitch[3] = <In4>;
multiswitch[4] = <In5>;
multiswitch[5] = <In6>;
multiswitch[6] = <In7>;
multiswitch[7] = <In8>;
if (pLocalEpics-><Sys>.MultiSwitch == 0) {
        for (ii=0; ii<8; ii++) multiswitch[ii] = 0.0;
}

<Out1> = multiswitch[0];
<Out2> = multiswitch[1];
<Out3> = multiswitch[2];
<Out4> = multiswitch[3];
<Out5> = multiswitch[4];
<Out6> = multiswitch[5];
<Out7> = multiswitch[6];
<Out8> = multiswitch[7];
```

7.7.6 RampSwitch



7.7.6.1 Function

The purpose of this block is to allow switching between two pairs of inputs.

7.7.6.2 Usage

This module passes a pair of inputs to the outputs, depending on the setting of the associated EPICS record.

7.7.6.3 Operation

If the associated EPICS record is equal to zero, Out1 will be set equal to In1 and Out2 will be set equal to In3. If the EPICS record is equal to one, Out1 will be set equal to In2 and Out2 will be set equal to In4.

7.7.6.4 Associated EPICS Records

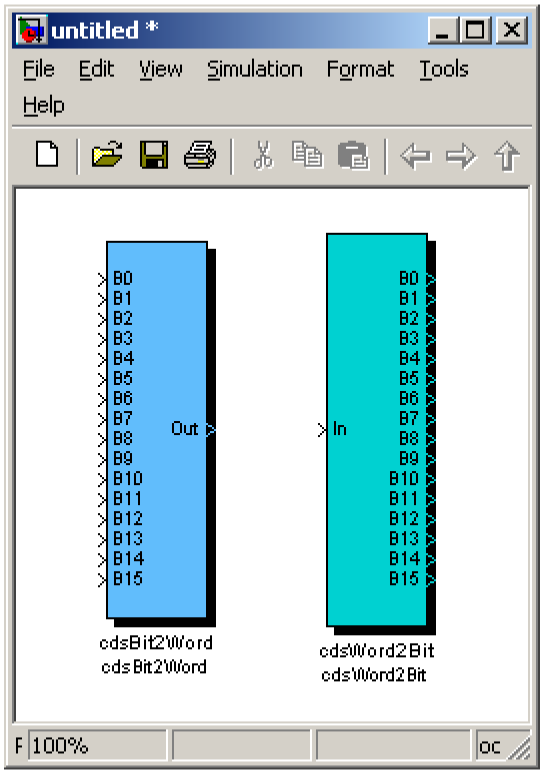
The RCG produces a single EPICS ‘bi’ record with the name given to this part by the user.

7.7.6.5 Code Example

The cdsRampSwitch module generates the following C code:

```
double rampswitch[4];

// RampSwitch
rampswitch[0] = <In1>;
rampswitch[1] = <In2>;
rampswitch[2] = <In3>;
rampswitch[3] = <In4>;
if (pLocalEpics-><Sys>.RAMPSWITCH == 0)
{
	rampswitch[1] = rampswitch[2];
}
else
{
	rampswitch[0] = rampswitch[1];
	rampswitch[1] = rampswitch[3];
}

<Out1> = rampswitch[0];
<Out2> = rampswitch[1];
```

7.7.7 cdsBit2Word/cdsWord2Bit



7.7.7.1 Function

The purpose of these two blocks is to convert from 16 single bit inputs to one 16-bit output word (cdsBit2Word) and from one 16-bit input word to 16 single bit outputs (cdsWord2Bit), respectively.

7.7.7.2 Usage

For cdsBit2Word, connect 16 binary inputs to ‘B0’ through ‘B15’, with the least significant bit connected to 'B0', the second least significant bit connected to ‘B1’, etc., and connect ‘Out’ to the module that should receive the 16-bit output word.

For cdsWord2Bit, connect the module that supplies the 16-bit input to ‘In’ and 16 binary outputs to ‘B0’ through ‘B15’, with the least significant bit connected to ‘B0’, the second least significant bit connected to ‘B1’, etc.

7.7.7.3 Operation

cdsBit2Word will calculate the output as Out = B0 * 1 + B1 * 2 + B2 * 4 + ... + B15 * 32,768 (i.e., Out = B0 * 2**0 + B1 * 2**1 + B2 * 2**2 + ... + B15 * 2**15), where B0 through B15 are equal to 1 or 0, e.g., if the binary inputs connected to B1, B2, B5, and B12 are equal to one and all other binary inputs are equal to zero, then the output (16-bit) word would be equal to (1 * 2 + 1 * 4 + 1 * 32 + 1 * 4,096 =) 4,134.

cdsWord2Bit will convert the 16-bit (integer) input, ‘In’, into 16 bits, e.g., the ‘In’ value 33,609 will result in the following bit pattern on the output: B15 = 1, B14 = 0, B13 = 0, B12 = 0, B11 = 0, B10 = 0, B9 = 1, B8 = 1, B7 = 0, B6 = 1, B5 = 0, B4 = 0, B3 = 1, B2 = 0, B1 = 0, and B0 = 1.

7.7.7.4 Associated EPICS Records

None.

7.7.7.5 Code Examples

The cdsBit2Word module generates the following C code:

```
int ii;

unsigned int cdsbit2word;
unsigned int powers_of_2[16] = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
                                1024, 2048, 4096, 8192, 16384, 32768};

if(feInit)
{

cdsbit2word = 0;

} else {



// Bit2Word
{
double ins[16] = {
        <B0>,
        <B1>,
        <B2>,
        <B3>,
        <B4>,
        <B5>,
        <B6>,
        <B7>,
        <B8>,
        <B9>,
        <B10>,
        <B11>,
        <B12>,
        <B13>,
        <B14>,
        <B15>
};
cdsbit2word = 0;
for (ii = 0; ii < 16; ii++)
{
if (ins[ii]) {
cdsbit2word += powers_of_2[ii];
}
}
}

<Out> = cdsbit2word;
```

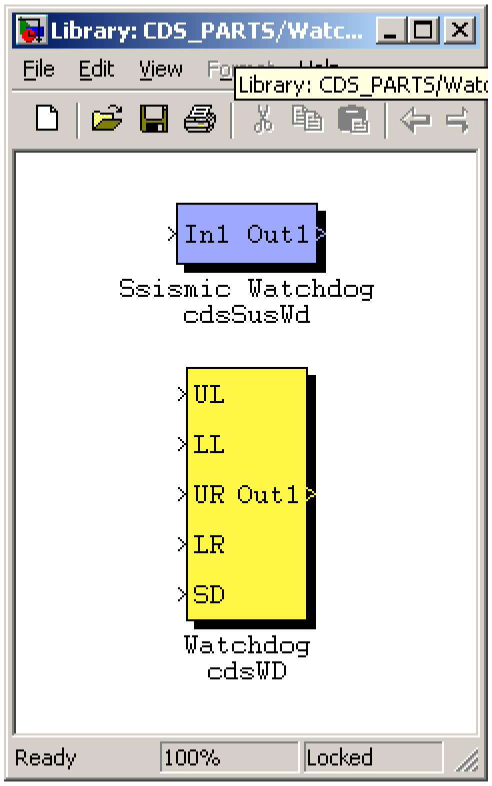
The cdsWord2Bit module generates the following C code:

```
int ii;

unsigned int cdsword2bit[16];

// Word2Bit
{
unsigned int in = (int) <In>;
for (ii = 0; ii < 16; ii++)
{
if (in%2) {
cdsword2bit[ii] = 1;
}
else {
cdsword2bit[ii] = 0;
}
in = in>>1;
}
}

<B0> = cdsword2bit[0];
<B1> = cdsword2bit[1];
<B10> = cdsword2bit[10];
<B11> = cdsword2bit[11];
<B12> = cdsword2bit[12];
<B13> = cdsword2bit[13];
<B14> = cdsword2bit[14];
<B15> = cdsword2bit[15];
<B2> = cdsword2bit[2];
<B3> = cdsword2bit[3];
<B4> = cdsword2bit[4];
<B5> = cdsword2bit[5];
<B6> = cdsword2bit[6];
<B7> = cdsword2bit[7];
<B8> = cdsword2bit[8];
<B9> = cdsword2bit[9];
```

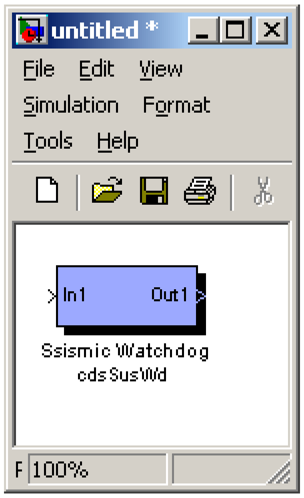
7.8 WatchDogs

Watchdogs are used to monitor their input signals and produce an error signal at their output to automatically trigger some fault handling code/modules. The modules to date were designed to implement similar tasks in initial LIGO controls.

NOTE: There is a third watchdog type (not shown), which was specifically implemented to replicate the watchdogs used in present LIGO HEPI systems. It is intended that it will be redesigned and added to the Watchdog parts library in a future release.



7.8.1 cdsSusWd



7.8.1.1 Function

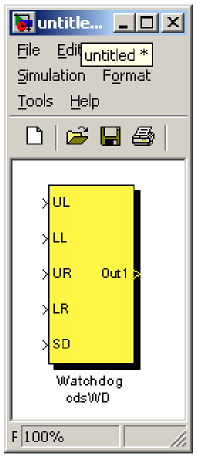
This function was developed with the sole purpose of connecting a suspension trip signal to the HEPI system in the early prototyping stages at LASTI (LIGO Advanced System Test Interferometer). This block should not be used in any new designs.

7.8.1.2 Usage

7.8.1.3 Operation

7.8.1.4 Associated EPICS Records

7.8.2 cdsWD

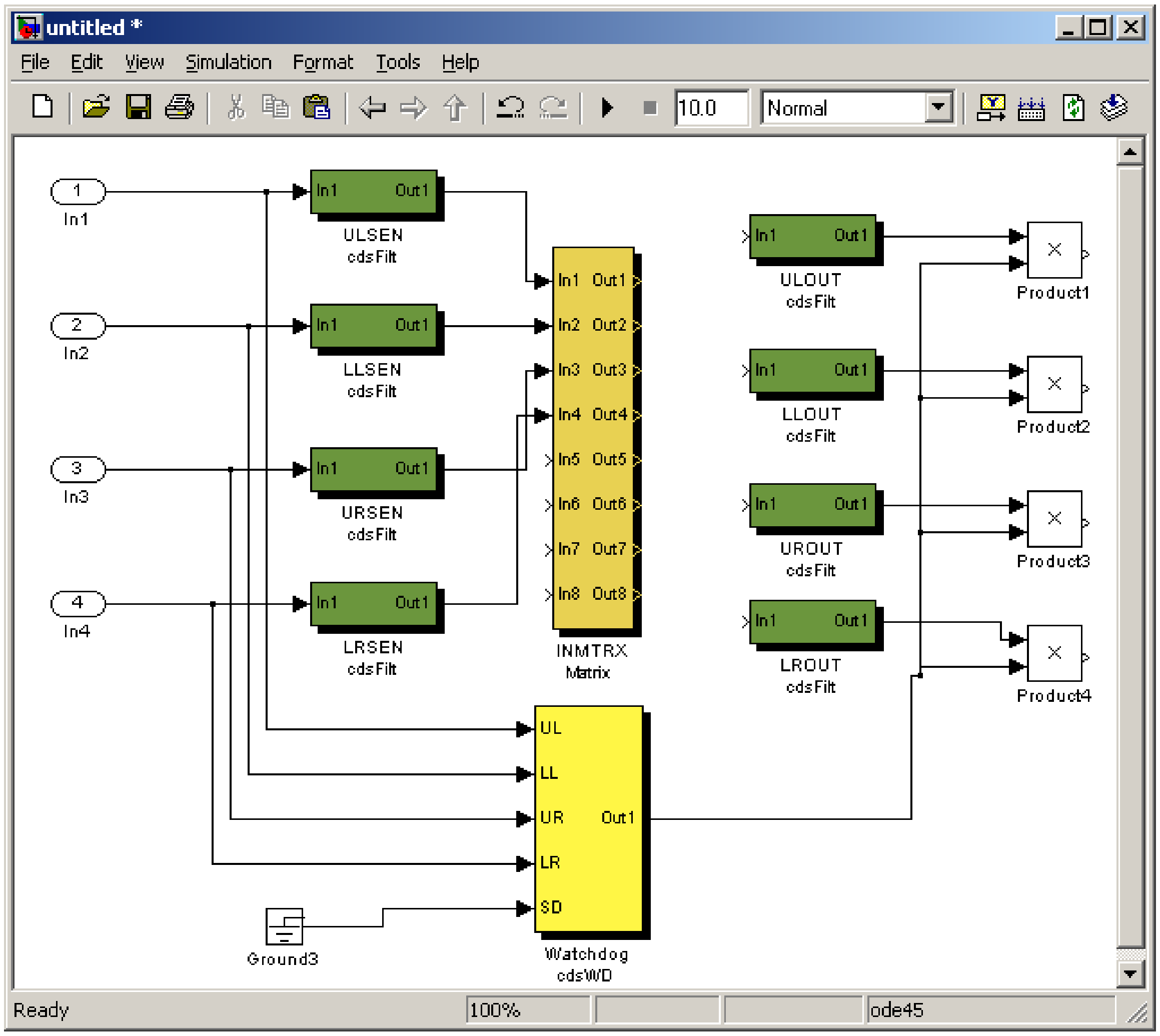


7.8.2.1 Function

This block was designed to implement the suspension watchdog function found in initial LIGO.

7.8.2.2 Usage

Typically, the raw suspension OSEM (Optical Sensing Electro-Magnet) signals are input at the left of the block. The output is then connected to a product block, with the second connection of the product being the signal path which is to be turned off if the watchdog trips. An (incomplete) example is shown in the following figure.



7.8.2.3 Operation

The run-time software for this module continuously calculates an RMS and variance for each input signal. If all variances are within the tolerances, the output is 1. If the variance for any input signal exceeds the RMS value beyond the operator set-points, the output becomes a value of 0, and remains 0 until reset by the operator.

7.8.2.4 Associated EPICS Records

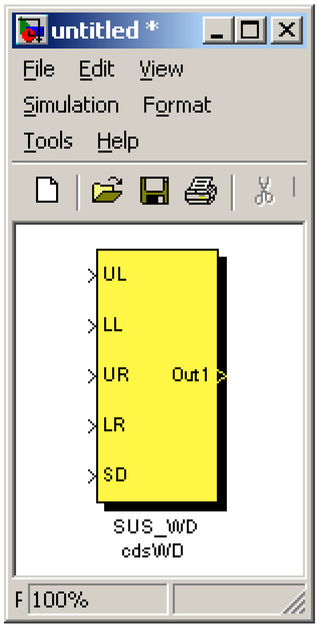
To support this module, the following EPICS records are produced for operator interaction. Signal names shown in the table are based on the part being named 'WD' in the user model.

Name	Type	Purpose
WD	Momentary ai	Used to turn the module on/off. If 'on', watchdog is operational. If 'off', the output of the watchdog code goes to 0. This is also used to 'reset' the watchdog after variances are back in tolerance.
WD_STAT	ai	Provides watchdog status information
WD_MAX	ai	Trip set-point. If variance on any input exceeds its RMS value by greater than this setting, the WD will trip.
WD_VAR_1 thru WD_VAR_5	ai	These records provide read-backs on the present variance of all five input signals.

7.8.2.5 Code Example

The cdsWD module generates the following C code:

```
int ii;

static int sus_wd;
static float sus_wd_avg[5];
static float sus_wd_var[5];
float sus_wd_vabs;

if(feInit)
{

sus_wd = 0;
for (ii=0; ii<5; ii++) {
        sus_wd_avg[ii] = 0.0;
        sus_wd_var[ii] = 0.0;
}
pLocalEpics-><Sys>.SUS_WD = 1;

} else {

// Wd (Watchdog) MODULE
if((clock16K % (FE_RATE/1024)) == 0) {
if (pLocalEpics-><Sys>.SUS_WD == 1) {
        sus_wd = 1;
        pLocalEpics-><Sys>.SUS_WD = 0;
};
```



```
double ins[5]= {
        <UL>,
        <LL>,
        <UR>,
        <LR>,
        <SD>,
};
   for(ii=0; ii<5;ii++) {
        sus_wd_avg[ii] = ins[ii] * .00005 + sus_wd_avg[ii] * 0.99995;
        sus_wd_vabs = ins[ii] - sus_wd_avg[ii];
        if(sus_wd_vabs < 0) sus_wd_vabs *= -1.0;
        sus_wd_var[ii] = sus_wd_vabs * .00005 + sus_wd_var[ii] * 0.99995;
        pLocalEpics-><Sys>.SUS_WD_VAR[ii] = sus_wd_var[ii];
        if(sus_wd_var[ii] > pLocalEpics-><Sys>.SUS_WD_MAX) sus_wd = 0;
   }
        pLocalEpics-><Sys>.SUS_WD_STAT = sus_wd;
}

<Out1> = sus_wd;
```