LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY

LIGO Laboratory / LIGO Scientific Collaboration

LIGO-T080135-v10 *LIGO* July 20, 2020

aLIGO CDS
Real-time Code Generator (RCG V4.0)
Application Developer's Guide

R. Bork

Distribution of this document:
LIGO Scientific Collaboration

This is an internal working note
of the LIGO Laboratory.

California Institute of Technology
LIGO Project – MS 18-34
1200 E. California Blvd.
Pasadena, CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project – NW22-295
185 Albany St
Cambridge, MA 02139
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

LIGO Hanford Observatory
P.O. Box 1970
Mail Stop S9-02
Richland WA 99352
Phone 509-372-8106
Fax 509-372-8137

LIGO Livingston Observatory
P.O. Box 940
Livingston, LA 70754
Phone 225-686-3100
Fax 225-686-7189

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 Advanced LIGO (aLIGO) 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.
- 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 T2000467: CDS RTS V4.0 Release Notes.
- LIGO T0900612 aLIGO CDS Design Overview https://dcc.ligo.org/LIGO-T0900612-v2 : Provides an overview of the aLIGO CDS hardware and software designs, along with links to more detailed documentation.
- LIGO T1000625 CDS Software Documentation https://dcc.ligo.org/LIGO-T1000625-x0: Provides links to this and other CDS software documentation.

4 RCG Overview

The RCG uses MATLAB Simulink as a 'drawing' tool to allow real-time control 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 Software Installation

As of CDS Real-Time Software (RTS) releave V4.0, the RTS software is available as Linux packages from the CDS advligoRTS gitlab site. See LIGO T2000467: CDS RTS V4.0 Release Notes for installation, configuration and runtime information.

4.2 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 RCG Software Parts Library.

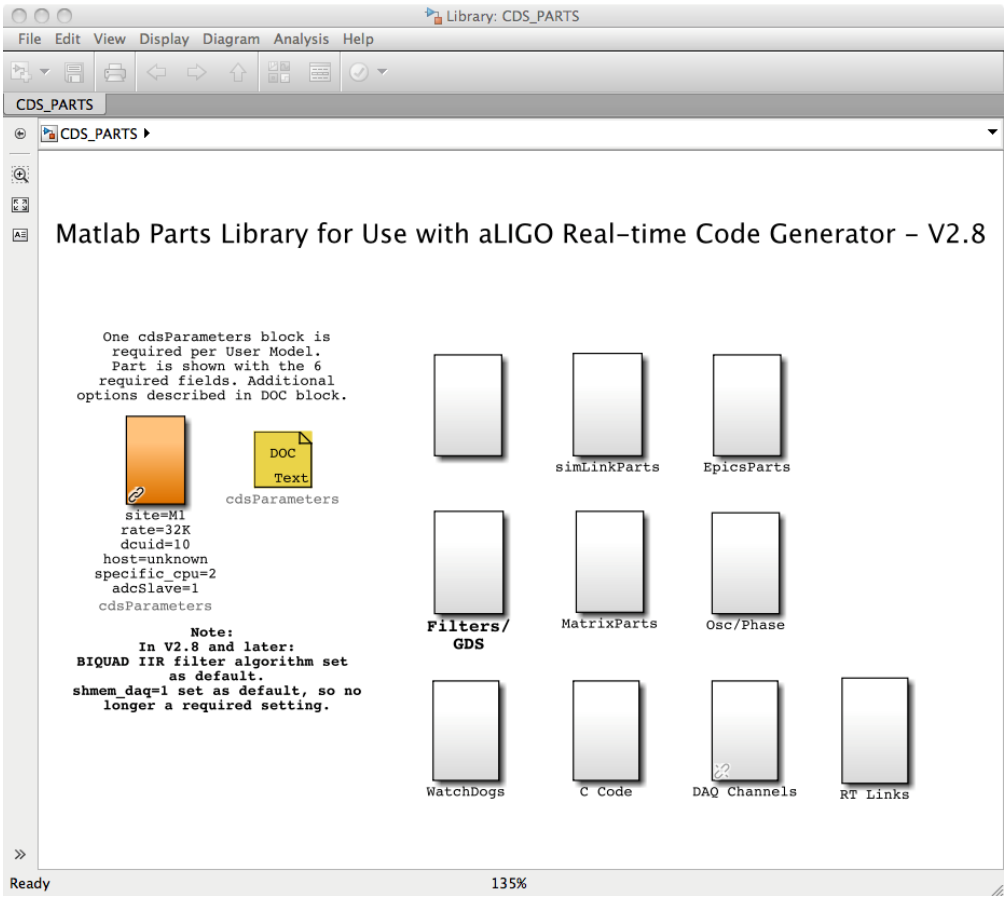


Figure 1: CDS Parts Library

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 figures, the first of which is the "top" level, the second showing the detail of one of the top level subsystem parts.

This example shows:

- A cdsParameters: This block must exist in all models. It is used by the RCG in setting code compile options and linking this application with various other components in a CDS distributed system.
- Three, 32 channel ADC (Analog-to-Digital Converter; ADC_0, ADC_1, ADC_2).
- One of each of the three supported DAC module.
- Within the subsystem level, selection of ADC channels and connection to CDS standard IIR filter modules.

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.

Many examples of models built for aLIGO use can be found within the CDS SVN repository (https://redoubt.ligo-wa.caltech.edu/websvn/) in the cds user apps area.

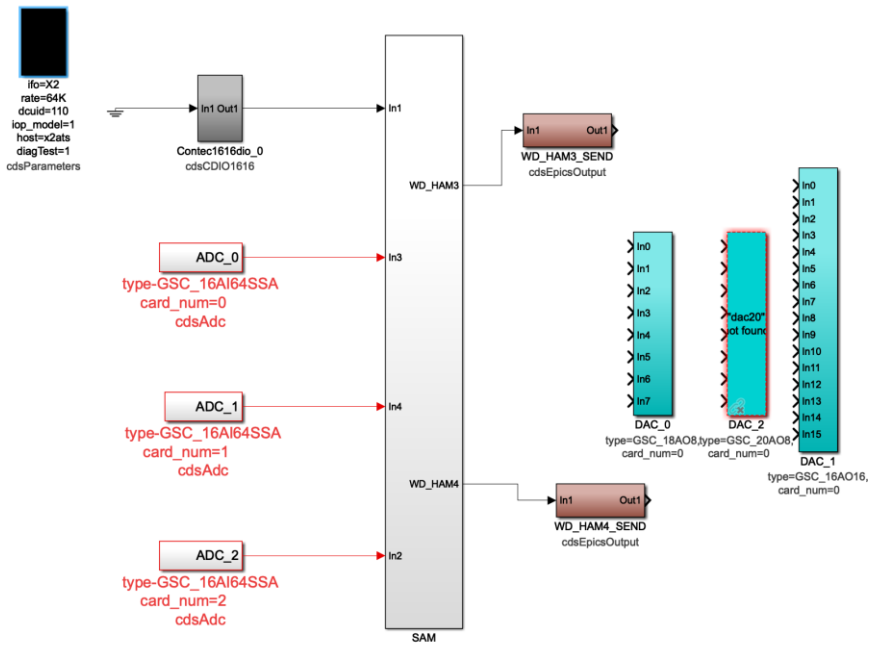


Figure 2: Example IOP Model - Top Level

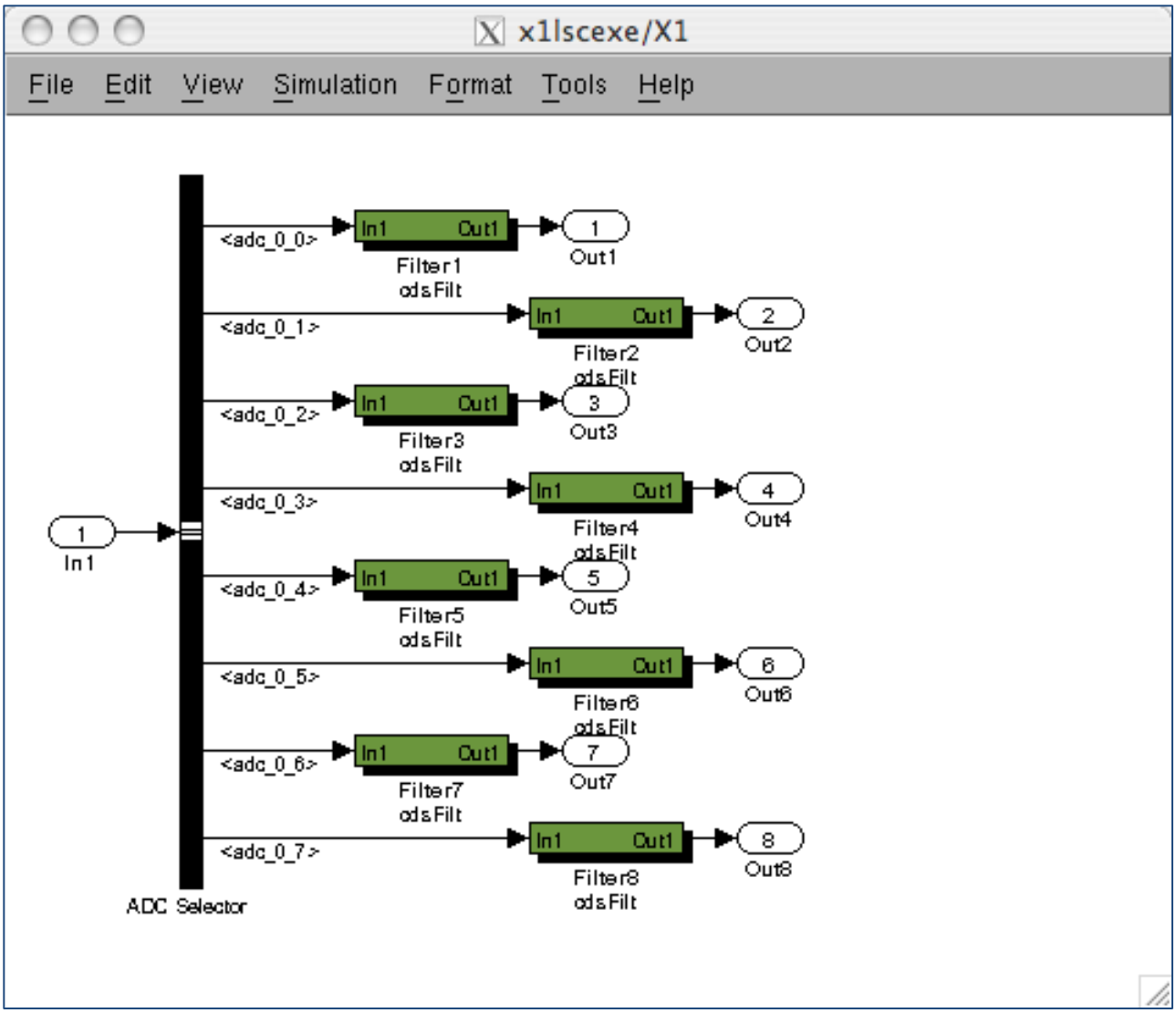


Figure 3: Example Model – Subsystem Level

4.3 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 in the designated CDS build directory. This results in the following actions:

a) A CDS Perl script (feCodeGen.pl) parses the user .mdl file and creates:

1) Real-time C source code for all of the parts in the user .mdl file, in the sequence specified by the links between parts.

2) A Makefile to compile the real-time C code.

3) A text file for use by a second Perl script to generate the EPICS code.

4) An EPICS code Makefile.

5) 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.

6) Reads/appends inter-process communications signals to an interferometer common text file.

b) 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.

c) The Perl script for EPICS code generation (fmseq.pl) is invoked, which:

1) Produces an EPICS database file.

2) 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.

3) Produces basic EPICS MEDM (Motif Editor & Display Manager) screens.

4) An EPICS BURT (Back Up and Restore Tool) back-up file for use in saving EPICS settings.

5) The header for the CDS standard filter module coefficient file.

6) A list of all test points, for use by the GDS (Global Diagnostic System) tools.

7) A basic DAQ (Data Acquisition) file.

8) A list of all EPICS channels for use by the EDCU (EPICS Data Collection Unit).

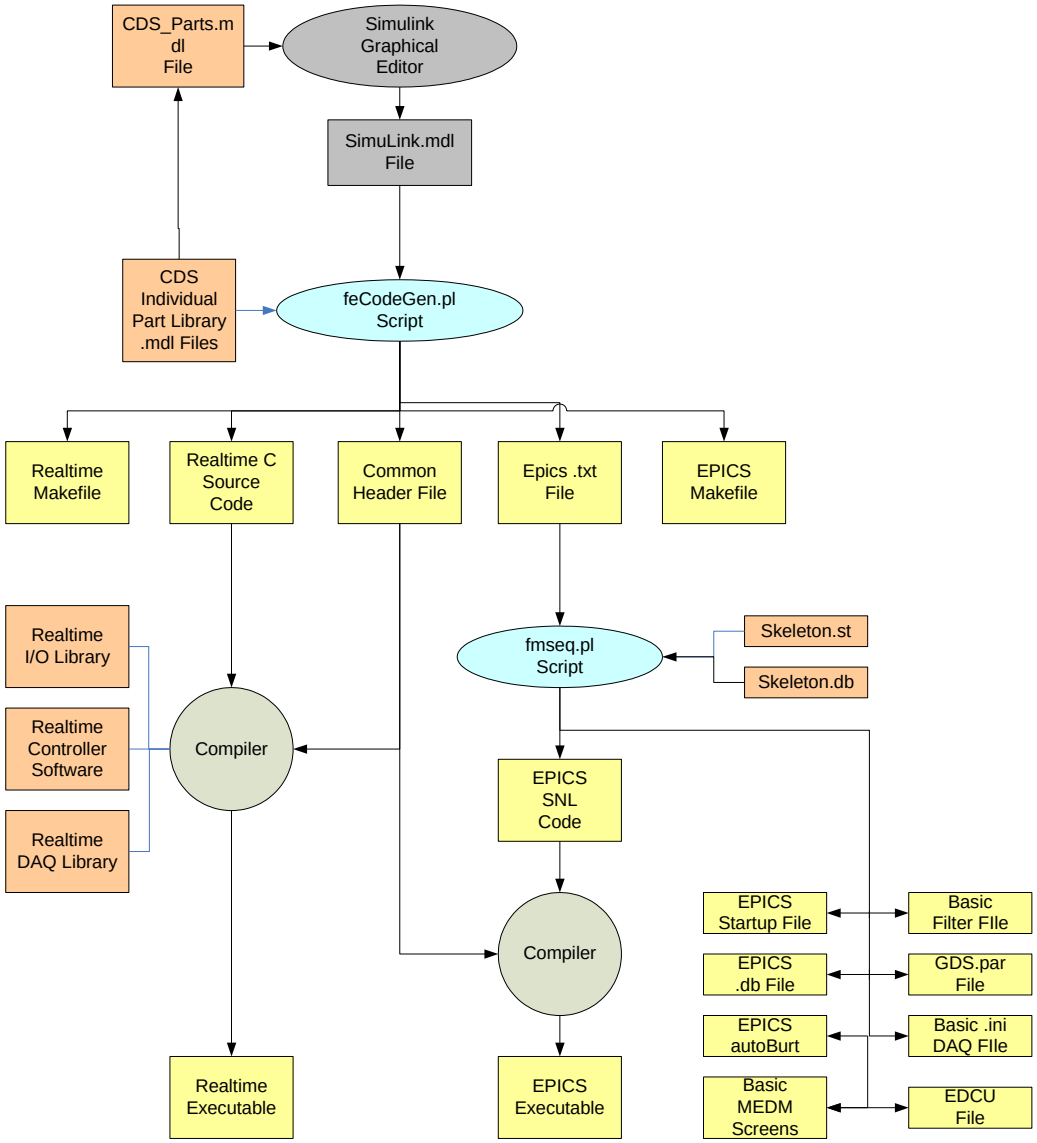


Figure 4: Code Generation

4.4 Run-time Software

The primary software modules that get executed on the CDS FE computers are shown in the figure below.

The computer itself is a multi-CPU and/or multi-core machine. The operating system is presently Debian 10 Linux, with a LIGO CDS custom patch for real-time applications. CDS applications are spread among the various CPU cores:

- CPU core 0: Reserved for the Linux OS and non-realtime critical applications.
- CPU core 1: Reserved for a special case RCG model known as in Input/Output Processor (IOP).
- CPU core 2 thru n: Real-time user applications built from the RCG to perform system control, referred to here as a Control Application Model (CAM). Any core not reserved for a real-time application is made available to the Linux OS to run non-realtime applications.

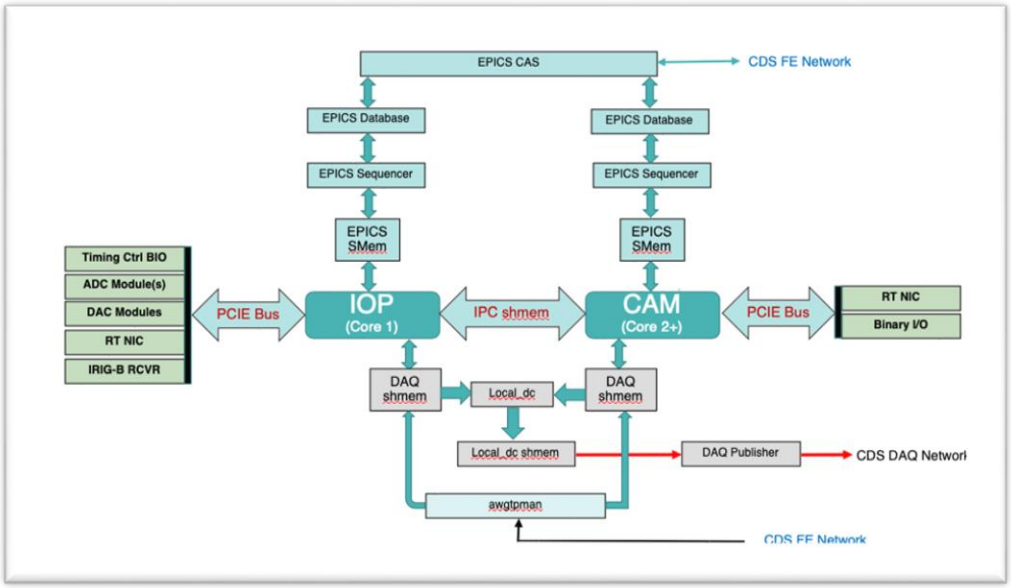


Figure 5: Run-time Software Overview

4.4.1 Real-Time

Each application built using the RCG from a Matlab model becomes a self-contained kernel module. At run time, it is loaded onto the CPU core specified in the model. This code makes use of the Linux OS facilities to load the code and allow the code to perform its necessary initialization. At that point, the code takes full control of the CPU core and that core is removed from the Linux list of available resources. This prevents that core from being interrupted and/or having other processes loaded by Linux. Code scheduling in now entirely controlled by the special case IOP software.

4.4.1.1 IOP

The IOP task is essentially the real-time scheduler for the FE computer. It is triggered by the arrival of data from the ADC modules, which are in turn slaved to the timing system (65536Hz clocks), which is locked to the GPS. It is also the conduit for passing ADC and DAC data between the PCIe modules and the user applications.

Key functions of the IOP include:

- Initialization and setup all PCIe I/O devices.
- Timing control, including:
    - Starting the clocks from the Timing receiver module in the I/O chassis such that startup begins synchronous with the GPS 1PPS mark.
    - Monitoring ADC data ready, caused by an ADC clock cycle, and initiating a real-time code cycle. This information is passed on to the user applications to synchronously trigger their code cycles.
- Synchronously reading ADC module data and passing data on to user applications.
- Synchronously writing data to DAC modules, data which is received from user applications.
- Providing real-time network and binary I/O module memory address information to user applications, such that these applications may communicate directly with those devices.

4.4.1.2 User Application

User applications are those that perform actual control functions. There may be as many user applications running on an FE computer as there are available cores (total cores – 2). Timing of these processes is controlled by the IOP and all ADC/DAC data is passed via the IOP to ensure synchronous read/write. The user applications may run at rates from 2K to 128K.

4.4.2 Non-Real-time

The 'Non-Real-time' CPU core(s) runs the following tasks:

- EPICS based network interface. This consists of several components:
    - Custom EPICS main.c process. This process monitors EPICS setpoint values and provides a number of reporting options. This monitoring is commonly referred to as SDF.
    - EPICS State Notation Language (SNL) sequencer software. This component is built and compiled by the RCG for each application. This code is designed to communicate data between the real-time application and the EPICS database records.
    - EPICS Database Records: Produced by the RCG and loaded at runtime. This EPICS database becomes the communication mechanism to various EPICS tools used in operating the system, via EPICS Channel Access (ECA). These tools include such items as MEDM, used to create and run operator interfaces.
- GDS Arbitrary Waveform Generator and Test Point Manager (awgtpman). For each real-time application, a copy of awgtpman is started. This program allows for the injection of test signals into the real-time application (AWG) and the readout of testpoint data, on demand, via the aLIGO DAQ system.
- Local DC (local_dc): New in V4.0 is a local data concentrator process. This software combines the DAQ data from all models running on an FE computer and writes this data to a local_dc shared memory block. At this point, the data is properly formatted for:
    - Connection to networking software to transmit the data to downstream DAQ computers.

  - Connect to a local copy of data acquisition software (daqd) for use on a standalone system.
- Data Publisher: In a distributed system, this software communicates DAQ data from real-time applications to the aLIGO DAQ system for archival and/or real-time diagnostic use. This process takes data from the local_dc shared memory and provides it on the DAQ from one or more data subscibers.

5 RCG Application Development

5.1 General Rules and Guidelines

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 that 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 48 characters (limit set by EPICS software). Underscores are used to further break up the name, with any number of characters in between.
3) The Matlab file name shall be of the form:
    a. IFO name (two characters eg h1.
    b. Subsystem name (three characters) eg sus, hpi, isi, etc.
    c. Remainder of name is arbitrary, but should provide a further description of the system to be controlled and must make the name unique for a particular installation.
    d. Examples for aLIGO: h1susetmx, h1susetmy, h1hpiham2. The RCG will pick off the first two characters as the interferometer (IFO) name and expect the next three characters to be the system name in order to produce a channel list consistent with (2) above.
4) Every model shall contain one, and only one, Parameter Block.
5) Every model shall contain at least one ADC part.

6) All I/O parts shall be on the model top level.
7) For ease of duplication, the top level of models should be limited to I/O parts, with other parts nested in subsystem components. For example, the following model could be duplicated by changing the “ETMX” subsystem block name to “ETMY” , change a few parameter block entries and I/O connections to make a new model to perform the same control functions on another suspension system.

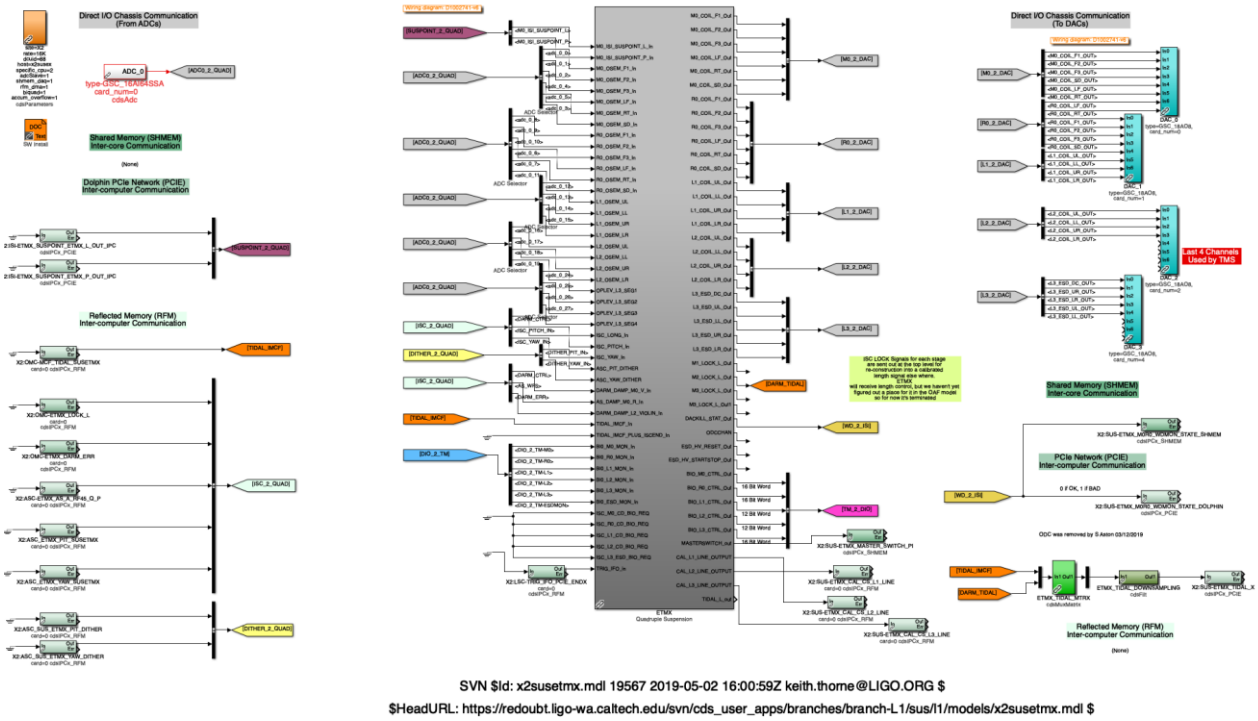


Figure 6: RCG Example

5.2 Code Compilation and Installation

In a standard aLIGO installation, a specific computer is set up by the site system administrator to compile user models. User models are controlled under the CDS SVN repository in the userapps area, with each major subsystem assigned a directory within this area. A new RCG user should contact the site system administrator for this information.

The CDS RTS software is now available for download as packages. This includes the RCG. There are now new build/install commands, covered under the next section, 5.2.1. For those pulling the RCG from the CDS GIT repository, the previous RCG compile options are also available, as described in Section 5.2.2.

5.2.1 Standard Compile and Install Using CDS RTS Package

For RCG 4.0, there are rtcds commands used to compile and install RCG software.

The commands to compile and install are:

1. Log into the site build machine.
2. Compile the application: rtcds build <modelname>
3. Install the code: rtcds install <modelname>

Note that it is no longer required to setup and go to a specific build directory as in previous releases.

5.2.2 Compile and Install Using RCG from GIT Repository

If RCG code is pulled from the GIT repository, then the previous RCG version 3 build procedures can still be used.

1. Pull code from GIT repository. Standard directory to install to is /opt/rtcds/rtscore/advligorts.
2. Create a build directory. Standard is to create a /opt/rtcds/site/ifo/rtbuild directory.
3. In the build directory, configure the build ie execute /opt/rtcds/rtscore/advligorts/configure.
4. Compile the control model ie make <modelnam>. Compilation products include:
    a. Real-time code source and executable kernel object in the *BUILD/src/fe/modelname* directory.
    b. EPICS database and compilation code in the *BUILD/build/modelnameepics* directory.
    c. Complete EPICS database and executable, ready for installation, in the *BUILD/target/modelnameepics* directory.
5. Install the code ie make install-<modelname>.

5.2.3 RCG Install Products

Using either the rtcds install or make install- commands described above results in the following code installation:

1) A complete backup of the previous code installation into the */opt/rtcds/site/ifo/target_archive/modelname* directory.
2) Autogenerated EPICS MEDM screens are moved into the */opt/rtcds/site/ifo/medm/modelname* directory. If the EPICS caQtDM package is installed, the RCG will also produce EPICS .ui files for use with caQtDM.
3) Runtime code moved into the */opt/rtcds/site/ifo/target/modelname* directory, including:
    - Real-time executable kernel object into the *bin* subdirectory.
    - EPICS related code and startup scripts into the *modelnameepics* subdirectory.

- Compilation information files into the *src* subdirectory. This area also contains a copy of all source code used in this build.
- On first compile of a model, produce a safe.snap file for use with the CDS setpoint monitoring software.

4) Appropriate GDS testpoint information moved into place for use by the DAQ and GDS software.
5) DAQ channel configuration file moved into /opt/rtcds/site/ifo/chans/daq directory. This file is used by the real-time code and DAQ system to acquire data.
6) IIR filter module coefficient definition file moved into /opt/rtcds/site/ifo/chans directory as MODELNAME.txt. This file is used by the **foton** tool to store filter coefficient information, loaded at run time by the real-time code to define its filter calculations. NOTE: The real-time code will also read FIR filter definitions from a separate file, if provided by the user ie not auto-generated and foton will not produce FIR filter coefficients. Also, the use of FIR filters is limited to polyphase FIR on systems that run only at 2048 or 4096 samples/sec.

6 Running the RCG Application

6.1 Application Startup

The RTS V4.0 and later software makes use of Linux systemd to control start/stop of RTS processes. There are two alternatives provided.

6.1.1 Using rtcds command

If the RTS packages were installed, then the following start/stop commands are available.

- Log on to the computer on which the code will execute.
- Start the model: rtcds start <modelname>
- Restart a running model: rtcds restart <modelname>
- Stop a running model: rtcds stop <modelnam>

Additional rtcds command line options can be found at rtcds command line interface.

6.1.2 Using systemctl command

- Log on to the computer on which the code will execute.
- Start the model: sudo systemctl start rts@modelname.target
- Restart a running model: sudo systemctl restart rts@modelname.target
- Stop a running model: sudo systemctl stop rts@modelname.target

6.2 Runtime Processes

6.2.1 Control Application Processes

The RTS startup commands will result in the loading and execution of the following processes:

- EPICS process (<modelname>epics). This includes:
  - Loading of the EPICS database.
  - Restoration of user defined “safe” startup settings from safe.snap

    - EPICS sequencer acting as interface between real-time kernel software and EPICS database.
    - Continuous monitoring of control setpoints.
        - Compares values in Setpoint Definition File (SDF) against runtime readings.
- Real-time kernel object produced by the RCG, which provides the actual control algorithms.
- The awgtpman process, to provide GDS support for this application.

6.2.2 DAQ Processes

6.2.2.1 Local DC

For both an FE in a distributed system or a CyMAC, the local_dc process must be started next.

local_dc -b <output_mbuf_name> -m 100 -s "system list" -w wait_time -d <par file path>

Where output_mbuf_name is the name of the output buffer to use. It defaults if not specified to "local_dc"

-m 100 is the size of the output buffer in MB. It can be [40-100]. It defaults to 100. This is the recommended size.

-s The system list. This is a space separated list of model names that the local_dc should pull data from. The list should not include the "_daq" suffix.

-d par file path. In order to properly handle test points the local_dc must determine the model rate (the rmIpcStr structure does not record that information). So the local_dc will try the following ways to determine rate information.

- If directory is specified with -d par files for the models will be searched for there.
- If the GDS_TP_DIR environment variable is set, that directory will be used to search for the par files.
- If the IFO and SITE environment variables are set the path to the par file will be created from those /opt/rtcds/SITE/IFO/target/gds/param/.
- Failing that the dcuid and rate (optional) can be encoded in the model name. This should only be done for something like the edc which is not a model. This is done by specifying the name as "model:dcuid:rate". So the edc's name would be specified as "edc:52:16" (or "edc:52" as the rate defaults to 16Hz if not specified).

6.2.2.2 DAQ Ethernet Connection

For an FE in a distributed system, the CDS DAQ network software must be started using the cps_xmit command. Cps_xmit is typically invoked as follows:

cps_xmit -b <<buffer_name> -m 100 -D <delay ms> -p <publisher_string>

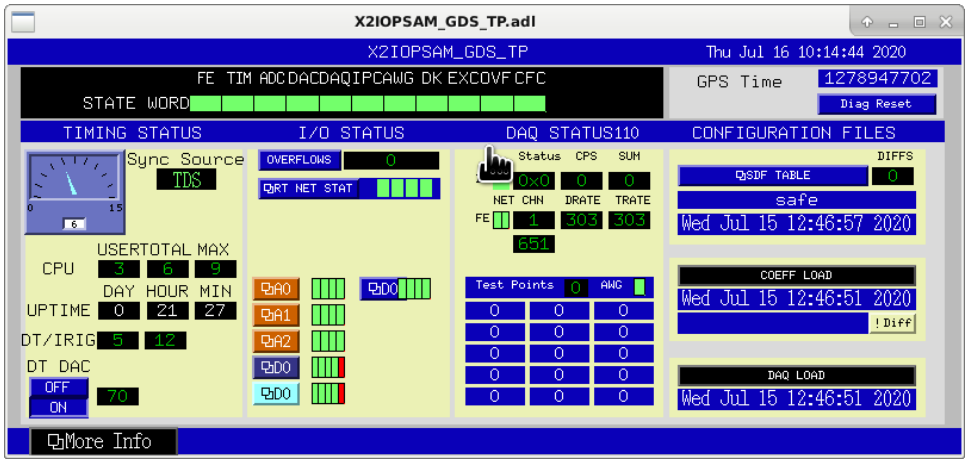
Where: -b is the buffer to read data from, it defaults to local_dc. -m is the size of the buffer in MB, it defaults to 100MB. -D is the delay in ms to apply before sending data. This is used to stagger the send times of the FE systems to reduce network collisions. -p is the publisher specification. It defaults to udp://127.0.0.1/127.255.255.255.

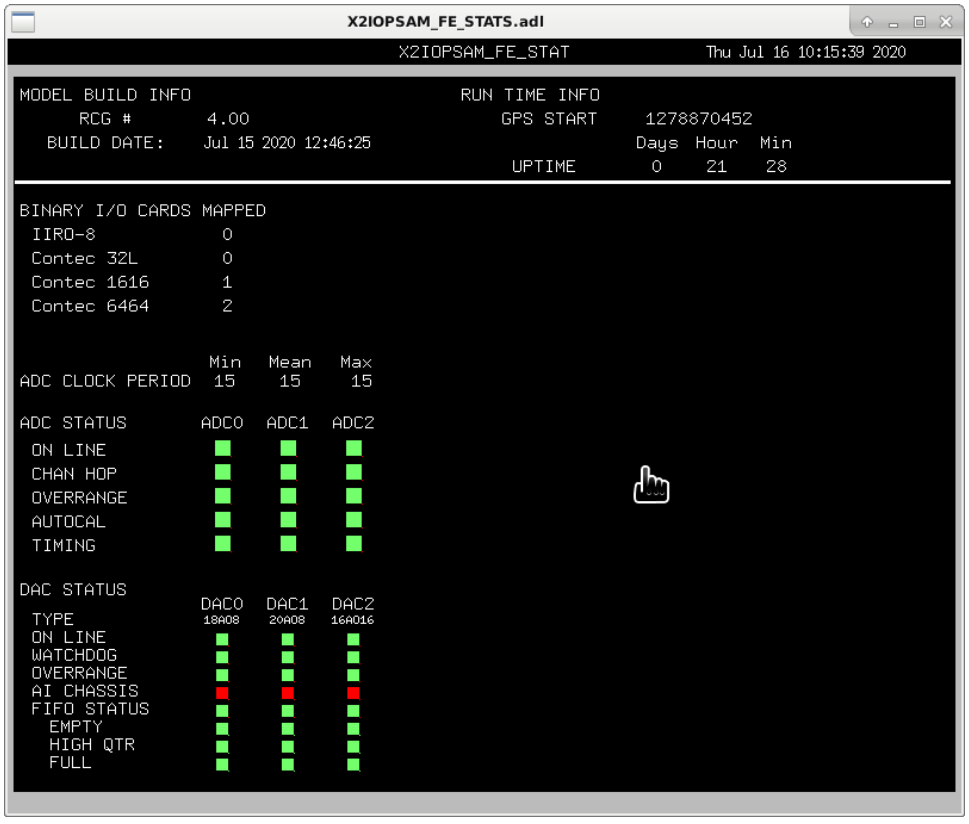
Cps_xmit currently supports 3 publishing modes:

- udp broadcast, specified by udp://<local iface name>/<broadcast address> As the name implies this does a udp broadcast with the code being able to retransmit data that was not received.
- tcp unicast, specified by tcp://<local ip address>:<port> This creates a service listening on the given address/port which will send out the data as 16Hz blocks of data is received.
- udp multicast, specified by multi://<local address>/<multicast address>:<port> As the name implies this does a udp multicast with the code being able to retransmit data that was not received. The implementation is presently limited to doing multicast over ipv4, and the local network.

6.3 Runtime Diagnostics

Once the code is running, a number of diagnostics, in the form of EPICS MEDM screens and log files, are available to verify proper operation. These diagnostics are described in LIGO-T1100625.





6.4 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.

- Setpoint monitoring tool: Part of every models EPICS task, used to monitor setpoints and alert operators when setpoints have been changed.
- EPICS StripTool: Provides strip charting for EPICS channels.
- Dataviewer: Allows users to view DAQ and GDS TP channels, either live or from disk.
- ndscope: A live “oscilloscope” type data display tool.
- Diagnostic Test Tool (DTT): Allows for analysis of live or recorded DAQ/TP data, particularly useful for calculating and plotting transfer functions.
- Foton: A GUI for the development of filter coefficients for use by the real-time software.
- Guardian: Python scripting tool used for supervisory level control automation.

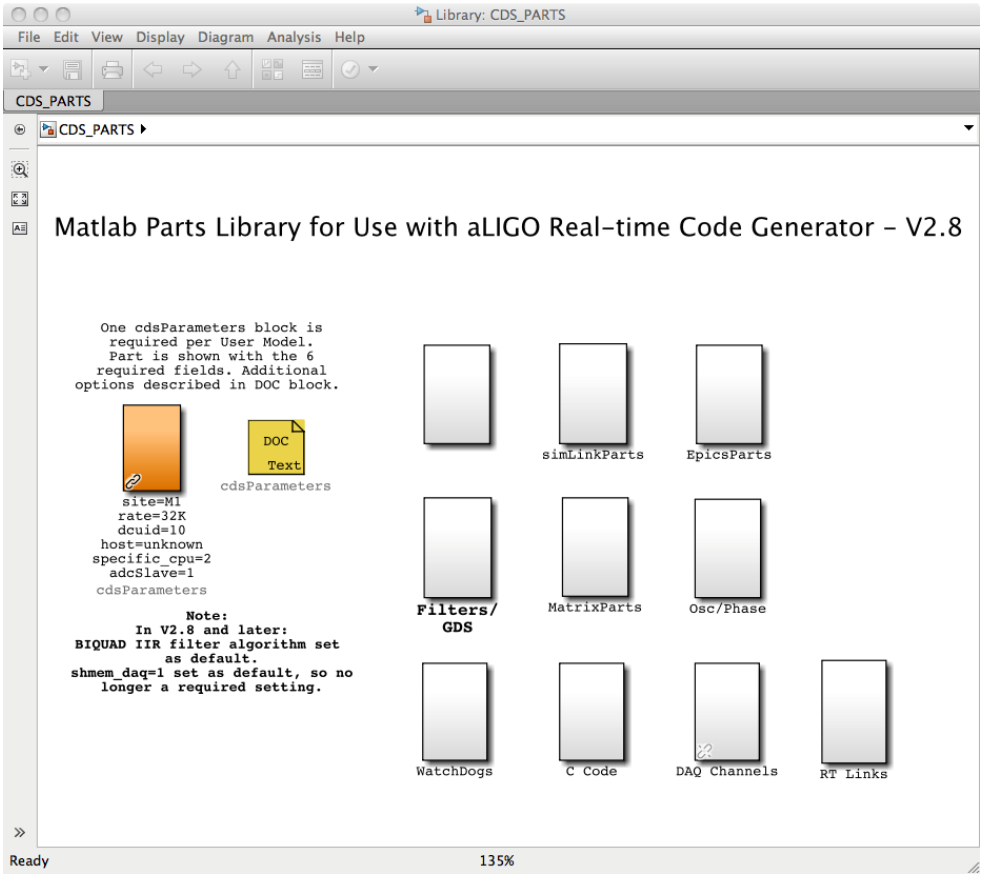
7 RCG Software Parts Library

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

CDS parts top level, shown below, contains:

1) Parameter Block: Required for all models.
2) Additional part subsystem blocks, which group parts by category.



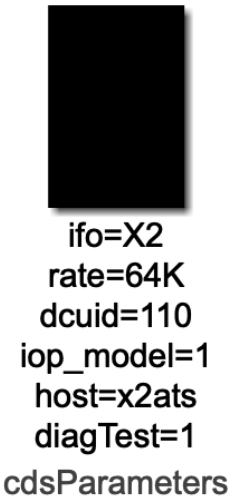
7.1.1 cdsParameters

7.1.1.1 Function

The RCG uses various entries in the control model Parameter block to properly configure the code produced by the RCG. The RCG supports two basic types of models:

- Input/Output Processor (IOP): Primary tasks are to map all the PCIe modules in the I/O Chassis (IOC), setup and maintain synchronization with the timing system and to provide the interface to ADC/DAC modules to control models running on the same computer. There must be one, and only one, IOP per front end (FE) computer.
- Control Application Model (CAM): These are user provided models intended to perform the actual real-time control and data acquisition functions. There may be up to (number of computer cores – 2) CAM loaded to a single FE computer. (Core 0 is reserved for Linux OS and core 1 reserved for IOP)

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. Based on information in this block, the RCG will produce either I/O Processor code (if iop_model=1) or Control Application Model (CAM) code (iop_model=1 does not appear in the parameter list).

7.1.1.2.1 IOP Parameter Block Settings

Some IOP settings have changed and a few added in V4.0 to accommodate various modes of operation. The following settings are standard for LIGO site production systems.

- The minimum required entries for all IOP configurations are:
  - ifo=x, where x is the ifo designator eg H1, L1, A1, etc.
    - Note: This replaces site=x, supported in previous releases. For V4.0, site= is still supported, but will be deprecated in future releases.
    - Note: A1 is to be the designator for LIGO India
  - iop_model=1: This indicates that the RCG should configure the code to be an IOP
    - Note: This replaces adcMaster=1. V4.0 still supports this, but this will be deprecated in future releases.
  - rate=x: The IOP code cycle rate. The standard rate for an IOP is 64K, matching the ADC/DAC clocks. There are several options, described in optional IOP configuration sections to follow.
  - host=x, where x is the name of the computer on which the code is to run.
  - dcuid=x, where x is a unique identifier number for data acquisition. This number must be in the range of 5 to 13 or 16 to 255.
- Standard options used in production systems:
  - pciRfm=1: Indicates FE computer should connect to the RTS real-time network for communications between FE computers.
  - dolphin_time_xmit=1: Indicates that this IOP should send its GPS time and cycle count out on the real-time network. This is used by IOP models on FE computers that do not have an I/O chassis connected for timing.
    - Note: There can be only one real-time network time transmitter.
  - requireIOcnt=1: Typically set for production systems. This will cause the IOP code to exit on startup if it does not find all the I/O modules specified in the IOP model. This is a safety factor to ensure the IOP does not read/write from the wrong ADC/DAC modules.

- dolphin_time_rcvr=1: Indicates that this IOP does not have an I/O chassis connected and will use time information from the time transmitter.
- rfm_delay=1: Typically used in end station to corner station IPC data transfers. Since the time of flight for signals over 4km can prevent these signals from arriving at the next code cycle of the receiver, data is shifted by one cycle (written ahead 1 cycle).

For V4.0, there are also various optional run modes for IOP models.

7.1.1.2.1.1 Long Range IOP Configuration

In standard operation, an IOP receives ADC data at 64K, performs its processing cycle, and waits for the next ADC data set to arrive. When a FE has its IOC connected over a long range fiber link, it may be necessary for the IOP to request multiple ADC data blocks before performing a process cycle due to data time of flight An example of this for LIGO is acquiring PEM data from mid-stations, where the IOC is located in the mid-station, 2km away from its FE computer located in the corner station.

To accommodate this, the IOP is setup to instruct the ADC modules to send two data sets every second (2^{nd}) clock cycle instead of a single set every clock cycle. Upon receipt of data, the IOP will run thru two code cycles, continuing to send/receive data to/from CAMs at 64KS/sec.

For this configuration, the necessary parameter block settings are:

- rate=32K
- clock_div=2

NOTE: The settings shown above are specific to the LIGO use case for the PEM mid station application and assumes a standard 64K ADC clock. These can be set differently, depending on the specific application. The rule for these settings is: **rate * clock_div = ADC clock frequency**

7.1.1.2.1.2 Running IOP at 128K

The RTS now supports running of models at 128K. This requires the following IOP parameter settings:

- rate=128K
- adcclock=128

7.1.1.2.1.3 Operation without a LIGO standard timing system

A LIGO standard timing system receiver is not always available, such as in lab spaces, and I/O timing clocks are provided by other means. The IOP code supports this via two methods:

- If a 1PPS signal is provided on the last channel of the first ADC, the IOP code will automatically choose this as the startup synchronization method.
- If no 1PPS synchronization signal is to be provided, then the IOP will perform startup synchronization based on the FE computer internal clock. This requires an additional parameter block setting:
  - no_sync=1

7.1.1.2.1.4 High speed ADC support

Support for the General Standards model 18AI32SSC1M PCIe ADC module has been added for. RTS V4.0. Use of this module is targeted at a specific A+ application. Future releases will have a finalized code set once requirements have been fully defined.

At this point, support is for the standard ADC clocking rates and a special 512KHz mode. For the 512KHz clock case, the IOP will acquire this data at 64K. This is done by having the ADC send 8 data sets every 8 clock cycles to the IOP. Upon receipt, the IOP will run through its model defined processing 8 times, once for each data set, and then wait for the next data set. So, effectively, the IOP is running at 512KHz and will send DAQ data at this full rate. (NOTE: Due to present per model data rate limitations, only one DAQ/TP channel can be selected).

In the present IOP code, it will still pass ADC data to CAMs at 64KS/sec. This is done by passing every 8th sample without decimation filtering (still awaiting requirements on this). If LIGO standard ADC modules (General Standards 16AI64SSC) are included in the IOP model, then the IOP code assumes these cards are being clocked at the standard 64K rate and read out and pass their data in the normal way. This does however require the IOC to be provided with two separate clocks, 512KHz and 64KHz.

NOTE: When there is a mix of 18AI32SSC and 16AI64SSC cards in the IOC, then a 16AI64SSC should be in the first ADC position on the bus. This allows for standard timing diagnostics.

To operate at the 512KS/sec mode, the parameter block settings required for the IOP are:

- rate=512K
- clock_div=8
- no_sync=1: Presently, there are no LIGO IOCs with 512K clock or multi rate clocks. Therefore, external clock(s) must be provided to the individual ADC modules.

7.1.1.2.1.5 Additional IOP Options

There are several additional options available for all IOP configurations.

- optimizeIO=1: Normally, the DAC FIFOs are initialized with two values and, as it runs, the IOP maintains the FIFO at this size. This does introduce a 1 to 2 cycle delay between when IOP writes data and DAC clocks data out. This is done to allow the IOP code to verify proper DAC clocking. With this option set, the IOP will not preload the DAC FIFOs. This will provide for a slightly lower phase delay of 1 to 2 cycles (15-30 usec, aassuming a 64K clock).
- no_zero_pad=1
    - Without this set, upsampling of data from lower application code rates to the 64K rate of DAC outputs uses a zero padding algorithm. This is the standard presently used in all aLIGO systems. As an example, if data needs to be upsampled by a factor of 8, the new value calculated for output by the application is applied to the upsample IIR filter to produce the first output sample. To produce the remaining 7 samples, zero is supplied to the to the IIR filter. This form up upsampling provides for a faster output response, however can have a down side of a noisy signal if a DC output level is desired.
    - With this parameter set, upsampling is done by repeatedly sending the application calculated output to the upsample IIR filter. This can have a slower output response, but will cleanly hold a DC level

7.1.1.2.2 CAM Parameter Block Settings

In RCG V4.0, if the parameter block does not contain iop_model=1, then the model is built as a CAM. Required CAM parameter block settings are:

- ifo=xy, where:
  - x is the SITE designator eg H for Hanford, L for Livingston, etc.
  - y is the IFO number for that site.
  - Note: This replaces site=xy, supported in previous releases. For V4.0, site= is still supported, but will be deprecated in future releases.
  - Note: A1 is to be the designator for LIGO India. (SITE=LAO, IFO=A1)
- rate=x: The CAM code cycle rate. Supported options are 2K, 4K, 16K, 32K, 64K and 128K.
- host=x, where x is the name of the computer on which the code is to run.
- dcuid=x, where x is a unique identifier number for data acquisition. This number must be in the range of 5 to 13 or 16 to 255.
- adcclock=128: Required only if the CAM is to run with an IOP running at 128K.

7.1.1.2.3 Parameter Block Options Common to both IOP and CAM

- ipc_rate=x, where x is 2048, 4096, 8192, 16384, or 32768 and x <= ½ the model rate. This parameter was added to reduce IPC data traffic in cases where the IPC transmitter is running much faster than the IPC receiver requires. A specific case is where SUS IOPs are sending watchdog IPC signals at 64K to SEI systems that only need the data at much lower rates.
  - **NOTE: This setting will affect all IPC data transmissions from the model ie is not per channel selectable**.
- no_cpu_shutdown=1: Normally, at runtime, the IOP kernel object will lock its designated core for its exclusive use. With this parameter set, the IOP will install itself as a standard Linux kernel module without core locking. This can be useful in software debugging and running the code on a computer with a standard Linux OS installed. Note that the code timing will not be entirely stable when run in this mode but should still be good enough for testing.
- accum_overflow=1:ADC overflow accumulator value. Without this flag set, ADC and DAC overrange counters get reset each second ie errors/sec. With this flag set, overrange indicators will count continuously until either the integer value limit is reached or the DIAG RESET has been set.
-

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.

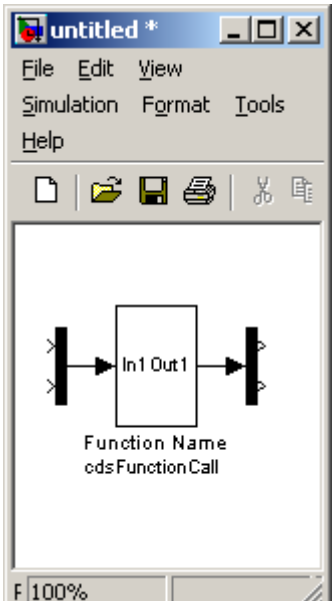
7.1.1.4 Associated EPICS Records

None.

7.2 C Code

The RCG provides the capability for application developers to provide their own C Code modules to be linked in with the real-time code build.

7.2.1 cdsFunctionCall



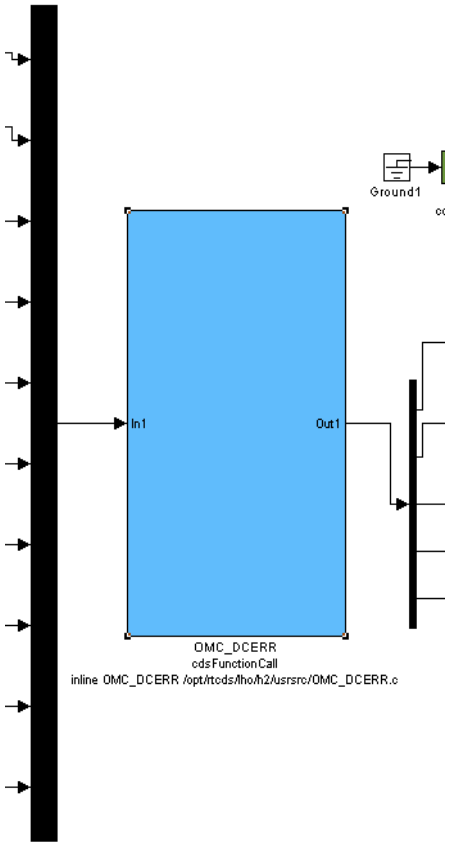
7.2.1.1 Function

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.

7.2.1.2 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.
- Block Properties must be modified to point to the code and its location in the form <inline> <C function name> <Source file>.
    - "inline" must be first entry, used as a flag to the compiler. This allows the same function to be used/called several times within a model and be provided with its own static variables.
    - C Function name, as defined in the source code file.
    - Source code file name. This can be either:
        - The complete path to the file, as in the example at right.
        - Environment variable + filename, for example $USER_CODE/omc_src.c, where $USER_CODE has been defined on the user's computer to point to the source code directory.



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

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

where:

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

7.2.1.3 Operation

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

7.2.1.4 Associated EPICS Records

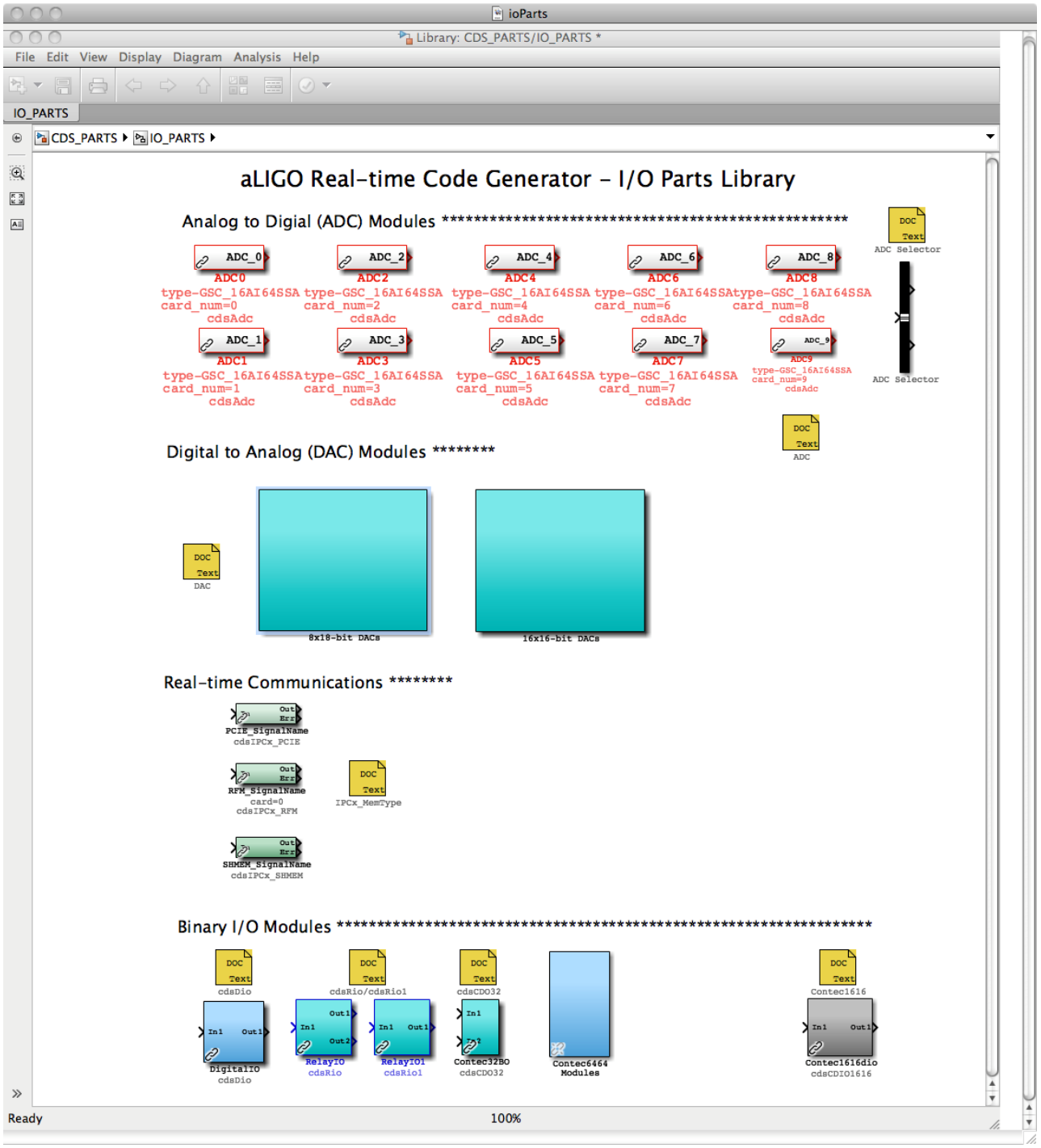
None.

7.2.1.5 Auto-Generated MEDM Screens

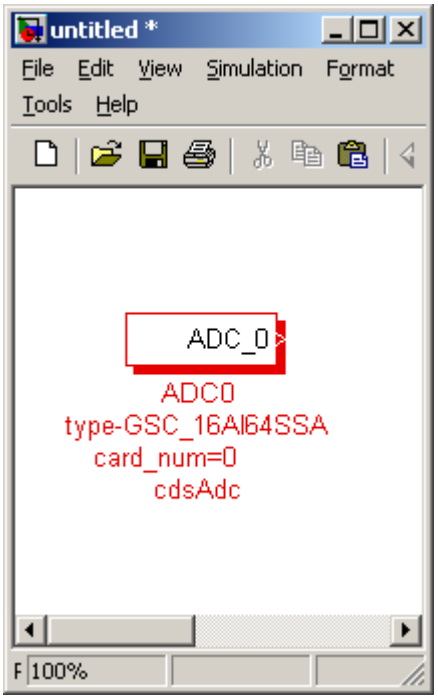
None.

7.3 I/O Parts

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



7.3.1 ADC



7.3.1.1 Function

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

7.3.1.2 Usage

Each RCG model must include at least one (1) ADC block. At models must start with ADC0, followed by ADC1, and so forth. The "card_num" should then be changed, as necessary, to point to the ADC module to connect to. A number of ADC blocks are available in the CDS_PARTS library for convenience, each with an embedded bus creator with pre-defined signal names. All

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.

7.3.1.3 Operation

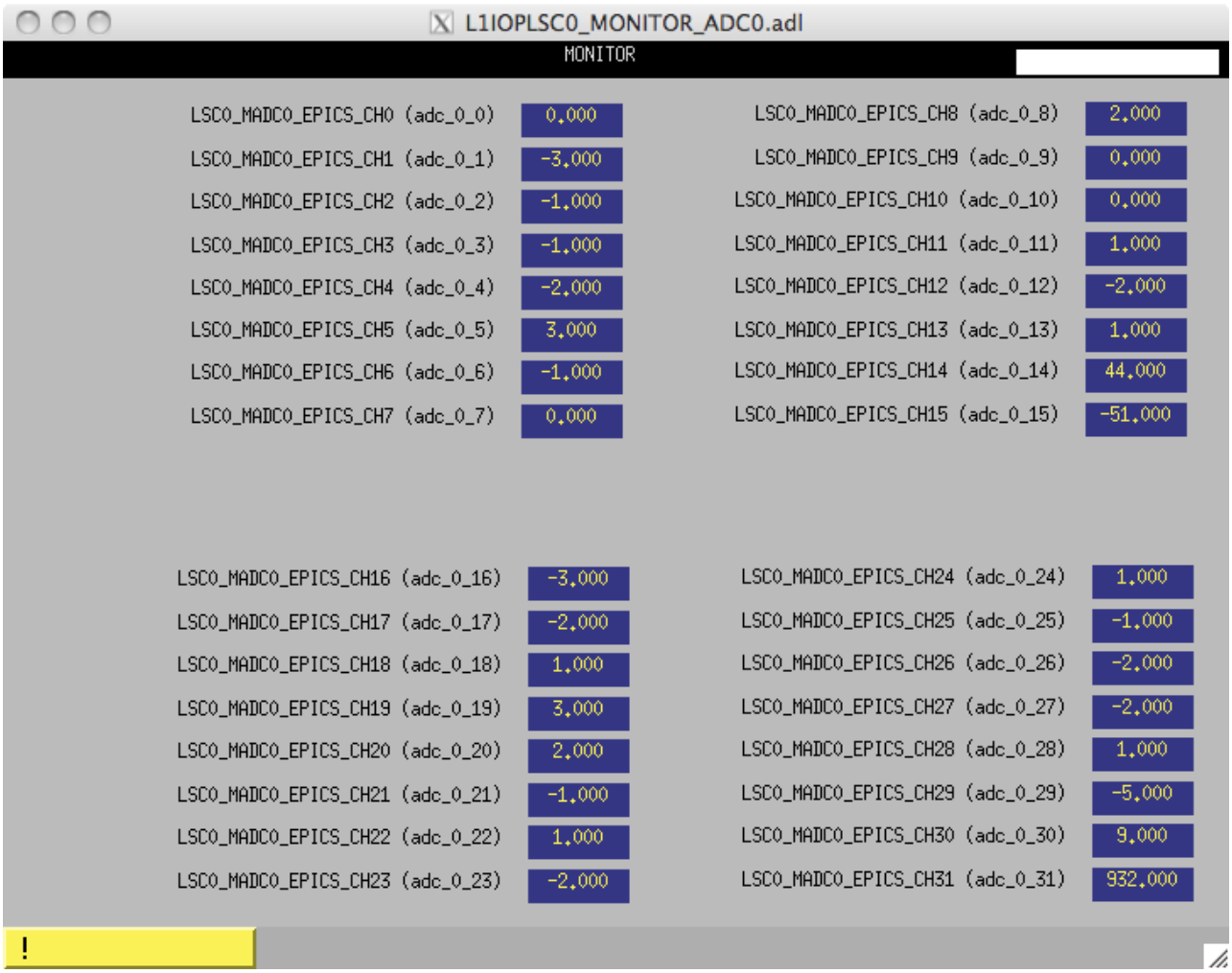
No software is directly produced for this part. Rather, it is used as an indicator of how many and of what type ADC module(s) the real-time I/O software should expect during operation.

7.3.1.4 Associated EPICS Records

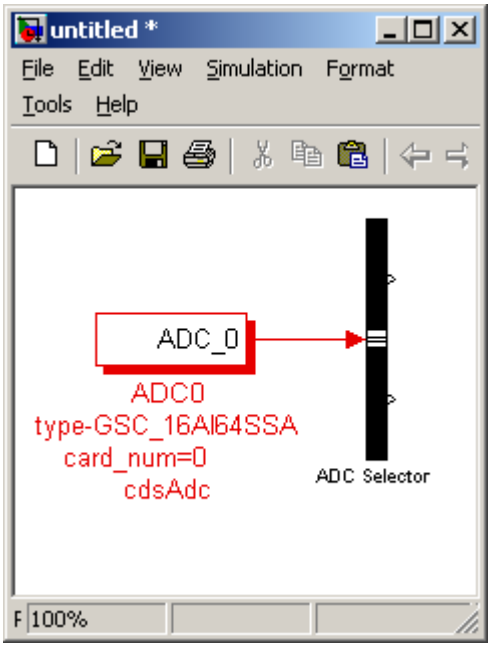
None.

7.3.1.5 Auto-Generated MEDM Screen

For each IOP and user application, a screen is created which shows raw ADC data input values. In the case of an IOP, this is the raw data received from the ADC module and being passed to user applications via shared memory. In the case of user applications, this is the data being received via the shared memory.


L1IOPLSC0_MONITOR_ADC0.adl
MONITOR
LSC0_MADC0_EPICS_CH0 (adc_0_0) 0.000
LSC0_MADC0_EPICS_CH1 (adc_0_1) -3.000
LSC0_MADC0_EPICS_CH2 (adc_0_2) -1.000
LSC0_MADC0_EPICS_CH3 (adc_0_3) -1.000
LSC0_MADC0_EPICS_CH4 (adc_0_4) -2.000
LSC0_MADC0_EPICS_CH5 (adc_0_5) 3.000
LSC0_MADC0_EPICS_CH6 (adc_0_6) -1.000
LSC0_MADC0_EPICS_CH7 (adc_0_7) 0.000
LSC0_MADC0_EPICS_CH8 (adc_0_8) 2.000
LSC0_MADC0_EPICS_CH9 (adc_0_9) 0.000
LSC0_MADC0_EPICS_CH10 (adc_0_10) 0.000
LSC0_MADC0_EPICS_CH11 (adc_0_11) 1.000
LSC0_MADC0_EPICS_CH12 (adc_0_12) -2.000
LSC0_MADC0_EPICS_CH13 (adc_0_13) 1.000
LSC0_MADC0_EPICS_CH14 (adc_0_14) 44.000
LSC0_MADC0_EPICS_CH15 (adc_0_15) -51.000
LSC0_MADC0_EPICS_CH16 (adc_0_16) -3.000
LSC0_MADC0_EPICS_CH17 (adc_0_17) -2.000
LSC0_MADC0_EPICS_CH18 (adc_0_18) 1.000
LSC0_MADC0_EPICS_CH19 (adc_0_19) 3.000
LSC0_MADC0_EPICS_CH20 (adc_0_20) 2.000
LSC0_MADC0_EPICS_CH21 (adc_0_21) -1.000
LSC0_MADC0_EPICS_CH22 (adc_0_22) 1.000
LSC0_MADC0_EPICS_CH23 (adc_0_23) -2.000
LSC0_MADC0_EPICS_CH24 (adc_0_24) 1.000
LSC0_MADC0_EPICS_CH25 (adc_0_25) -1.000
LSC0_MADC0_EPICS_CH26 (adc_0_26) -2.000
LSC0_MADC0_EPICS_CH27 (adc_0_27) -2.000
LSC0_MADC0_EPICS_CH28 (adc_0_28) 1.000
LSC0_MADC0_EPICS_CH29 (adc_0_29) -5.000
LSC0_MADC0_EPICS_CH30 (adc_0_30) 9.000
LSC0_MADC0_EPICS_CH31 (adc_0_31) 932.000
!

7.3.2 ADC Selector



7.3.2.1 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).

7.3.2.2 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.

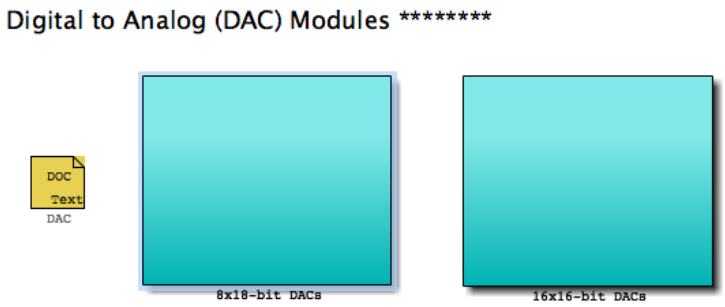
7.3.2.3 Operation

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

7.3.2.4 Associated EPICS Records

None.

7.3.3 DAC Modules



7.3.3.1 Function

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

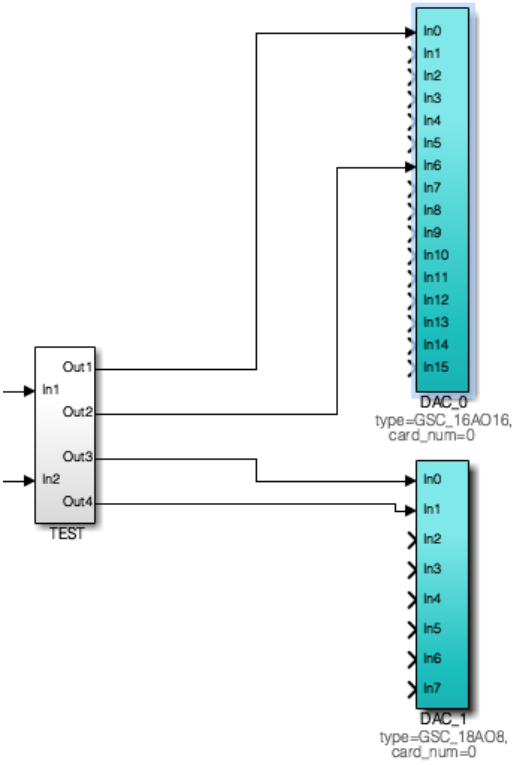
7.3.3.2 Usage

Two type of DAC modules are supported:

1) 16 Channel, 16 bit from General Standards.
2) 8 Channel, 18 bit from General Standards.

To use:

1) Drag and drop the appropriate model to the user model.
2) Change the part name to reflect instance of DAC part within the model. As with ADC parts, first DAC part must be named “DAC_0” and then number ending must increment by one for each DAC module used.
3) Use the block properties to select the desired DAC within the I/O chassis to connect to.



7.3.3.3 Operation

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

7.3.3.4 Associated EPICS Records

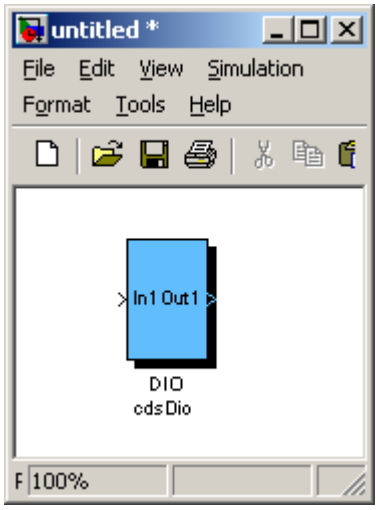
None.

7.3.3.5 Auto-Generated MEDM Screens

This display shows four (4) DAC modules, with two columns each:

1) Left is value being sent (OUT):
   a. For IOP, actual value it is sending out to the DAC module. The Red/Green indicator above this column indicates whether or not the IOP is receiving synchronous data from a user application to send out to the DAC. This indicator will go RED and output discontinued if there is not an application running, or running properly, to send data to the DAC eg user application is stopped.
   b. For user app, actual value being sent to shared memory for IOP to relay to DAC module.
2) Right is overflow counter (OFC) ie number of times per second output value exceeds +/- 32000 counts (16 bit DAC) or 128000 counts (18 bit DAC).

7.3.4 cdsDio



7.3.4.1 Function

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

7.3.4.2 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.

7.3.4.3 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.

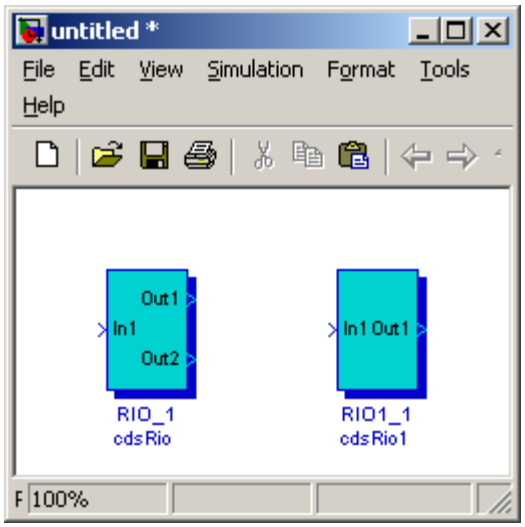
7.3.4.4 Associated EPICS Records

None.

7.3.4.5 Auto-Generated MEDM Screens

None.

7.3.5 cdsRio and cdsRio1 –



7.3.5.1 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.

7.3.5.2 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.

7.3.5.3 Operation

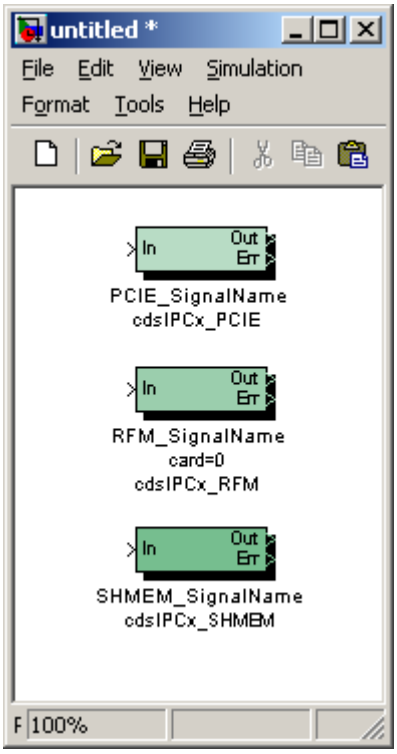
7.3.5.4 Associated EPICS Records

None.

7.3.5.5 Auto-Generated MEDM Screens

None.

7.3.6 cdsIPCx_PCIE, cdsIPCx_RFM, and cdsIPCx_SHMEM



7.3.6.1 Function

The purpose of these modules is to allow inter-process communications (IPC), via a PCI Express (PCIE) Network or via a Reflected Memory (RFM) Network for applications running on different computers or via Shared Memory (SHMEM) for real-time processes running on the same computer (but on separate CPU cores). These modules supersede the cdsIPCx module.

7.3.6.2 Usage

The user must change the label to a signal name of the following format (e.g.): H1:LSC-READOUT, where 'H1' is the IFO id. and the part following the colon is a unique identifier for this particular Inter-Process Communications (IPCx_<mmm>) module.

7.3.6.3 Operation

A separate IPC parameter file is maintained for each interferometer (IFO). This file is located in the /opt/rtcds/<site>/<ifo>/chans/ipc directory and its name must be <IFO>.ipc (e.g., H1.ipc). This file must include a (five or more lines) data record for each IPCx module being used. The first line should give the signal name (in all upper case) enclosed in square brackets. The second line should give the IPC communication mechanism (SHMEM for Shared Memory, RFM for Reflected Memory Network, or PCIE for PCI Express Network) in the format 'ipcType=<communication mechanism>'. The third line should give the sender data rate in the format 'ipcRate=<data rate>. The fourth line should give the host name in the format 'ipcHost=<host name>'. The fifth line should give the the IPCx Number in the format 'ipcNum=<number>'. This can be followed by one (or several) comment line(s), either beginning with 'desc=' or beginning with a '#' sign (and followed by a comment or descriptive text). The entries in this file can either be generated manually or be generated automatically (during the make process). Please note that automatic IPC entry generation is only possible for SENDER modules, i.e., the make process must be repeated until all modules (both SENDER and RECEIVER type) have been processed in two or more user models where all included IPCx modules are used as SENDERs.

A SENDER module is defined by having a signal attached to its input, but NO signal attached to its outputs ('Out' and 'Err'). A RECEIVER module is defined by having a "Ground" attached to its input and the output signal attached to some other module (e.g., and EPICS output module, a Filter module, etc.). The 'Err' output is only defined for RECEIVER modules and it can either be attached to some other module or be left un-attached.

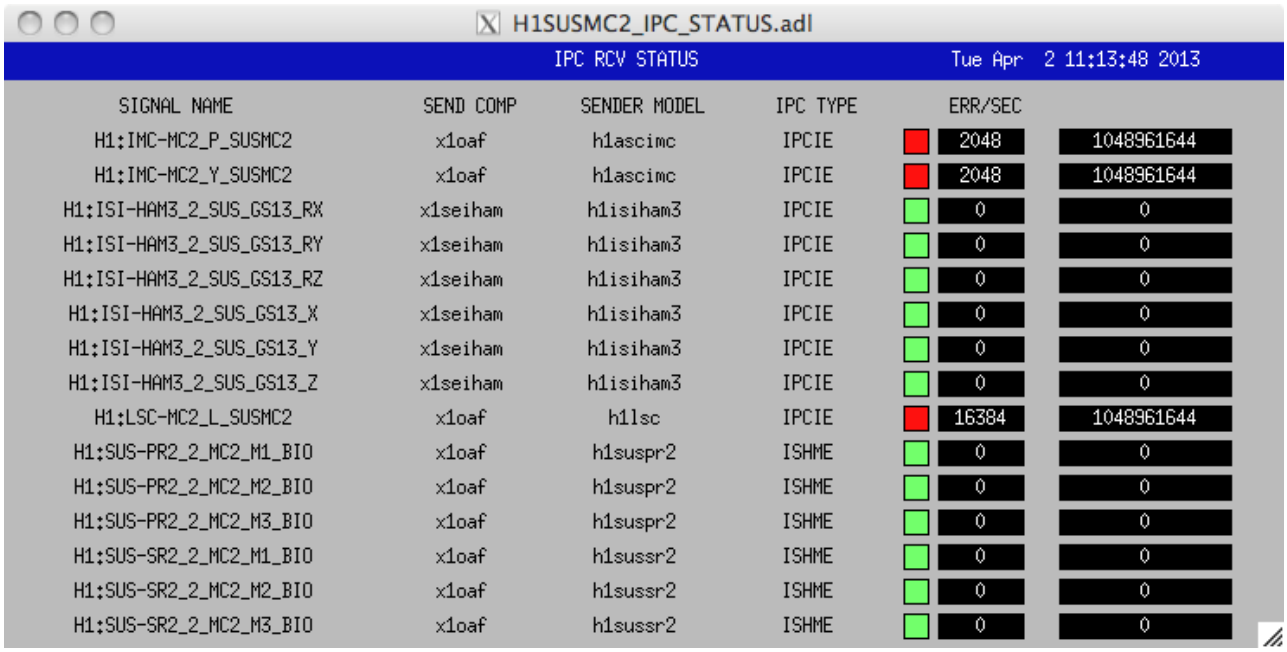
7.3.6.4 Associated EPICS Records

None.

7.3.6.5 Auto-Generated MEDM Screens

An IPC Status screen is generated for each RCG code model. An example is shown below. Information includes:

- SIGNAL NAME: Name of the signal being received.
- SEND COMP: Name of the computer from which signal is being sent.
- SENDER MODEL: Name of the control model from which signal is being sent.
- IPC TYPE: Communication mechanism/network.
- STATUS: RED/GREEN indicator of IPC faults. Upon detection of fault, this indicator remains latched RED until "DIAG RESET" is pushed on GDS_TP screen.
- ERR/SEC: Errors detected per second. If errors are continuing, this field will update every second. If errors have stopped, number will be latched, as with STATUS above.
- ERR TIME: GPS time of the last error detection. If errors are continuing, this field will update every second. If errors have stopped, number will be latched, as with STATUS above.



7.3.7 cdsCDO32

7.3.7.1 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.

7.3.7.2 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.

7.3.7.3 Associated EPICS Records

None.

7.3.7.4 Auto-Generated MEDM Screen

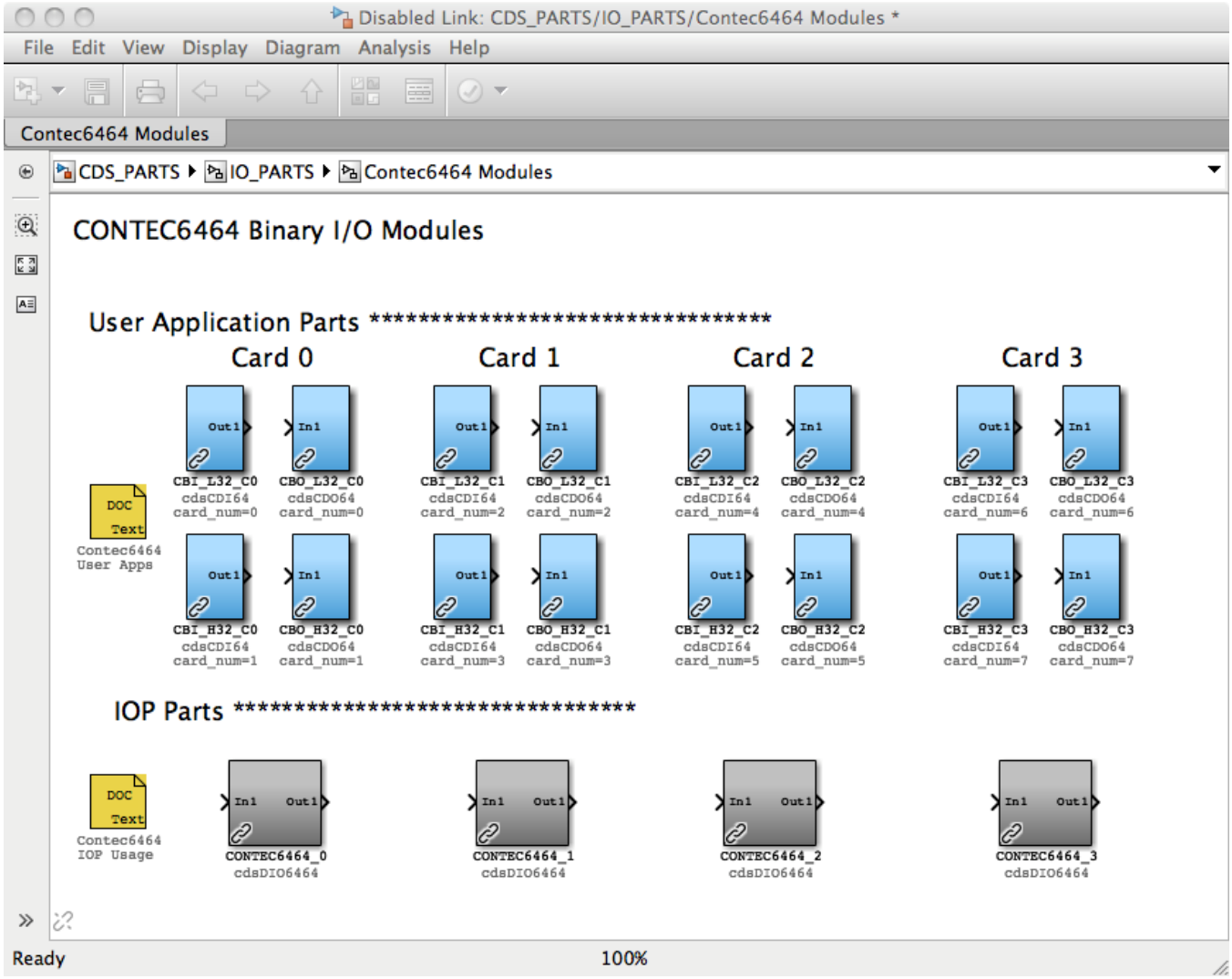
None.

7.3.8 cdsCDIO1616 and cdsDIO6464

7.3.8.1 Function

Used to connect Contec binary I/O modules.

NOTE: cdsDCIO1616 is designed only for use in an IOP to control the timing system.



7.3.8.2 Usage

The Contec1616 part should only be used in an IOP model. No input/output connections are required.

Use of the Contec6464 is used differently in an IOP than a user application model:

- In an IOP model, there should be one instance of a Contec6464 part for each card of that type in the I/O chassis. The NAME field should end in the card instance number, for example DIO_0, DIO_1, etc.
- Because of the large number of bits in this module and having to pass all of these as significant bits to the EPICS interface, each Contec6464 card defined in the IOP is presented as two 32 bit devices on the user side (lower 32 bit read/write and upper 32 bit read/write registers). Therefore, part naming is different on the user application model side. For example, if the user model is to address the lower 32 bits of the first Contec6464

card in the I/O chassis, the NAME field must end in _0. To access the upper 32 read/write bits, the NAME field must be end in _1.

7.3.8.3 Operation

Values from the card are read once per second.

Outputs are written whenever the value at the input to this part changes.

7.3.8.4 Associated EPICS Records

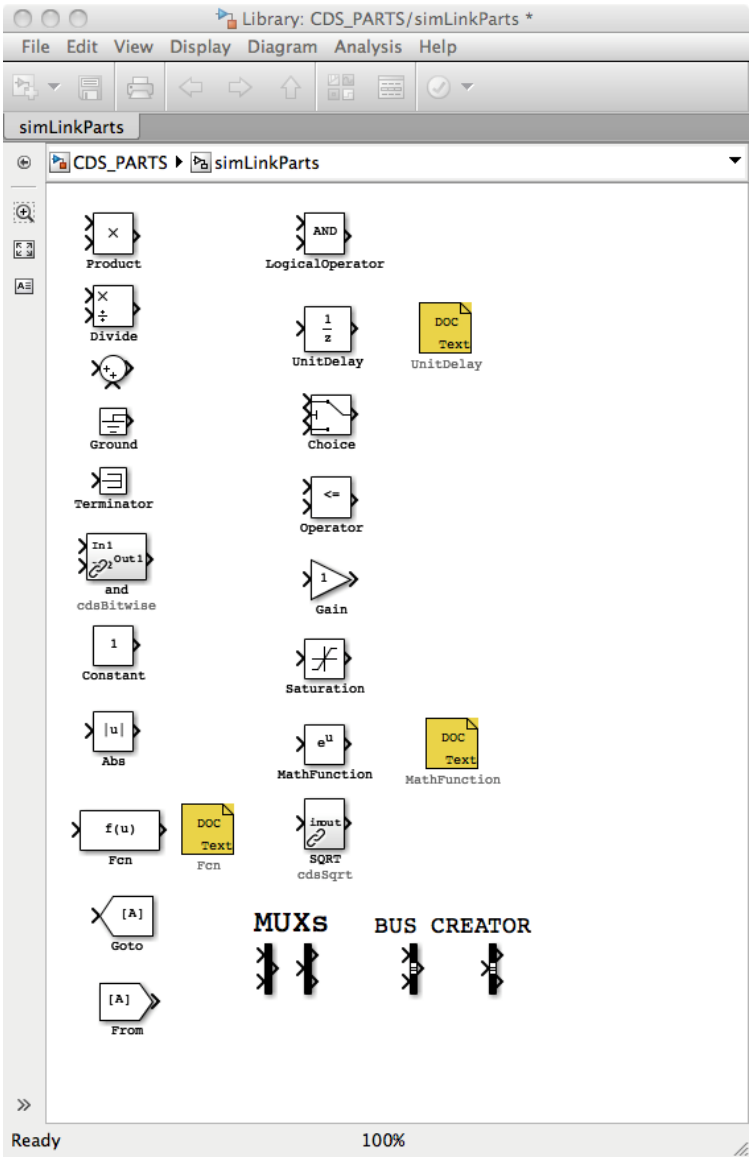
None.

7.3.8.5 Auto-Generated MEDM Screen

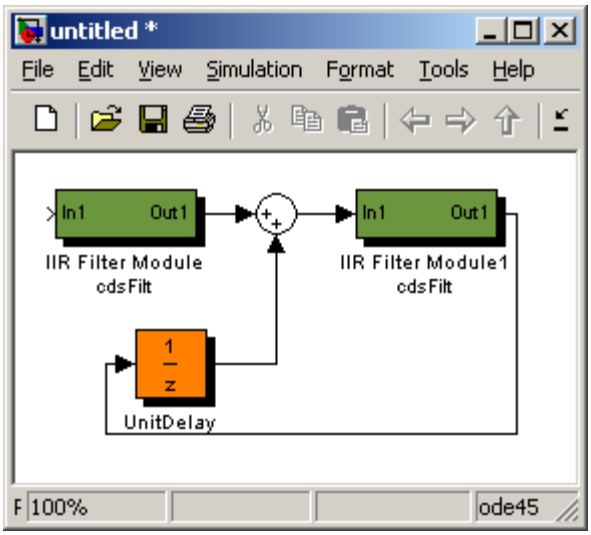
None.

7.4 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.



7.4.1 Unit Delay



7.4.1.1 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

7.4.1.2 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.

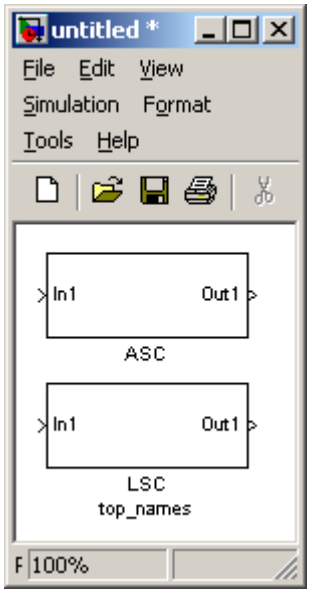
7.4.1.3 Operation

Introduces a one cycle delay between input and output.

7.4.1.4 Associated EPICS Records

None.

7.4.2 Subsystem Part



7.4.2.1 Function

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

7.4.2.2 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.

7.4.2.3 Operation

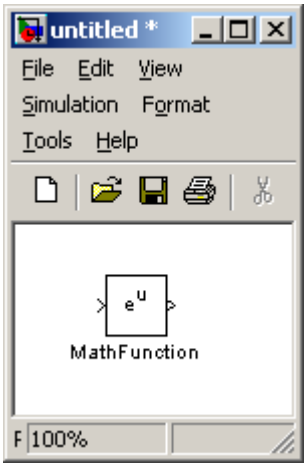
The subsystem part is only used by the RCG to produce appropriate signal names.

7.4.2.4 Associated EPICS Records

None.

7.4.3 MathFunction

7.4.3.1 Function



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

7.4.3.2 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.

7.4.3.3 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:

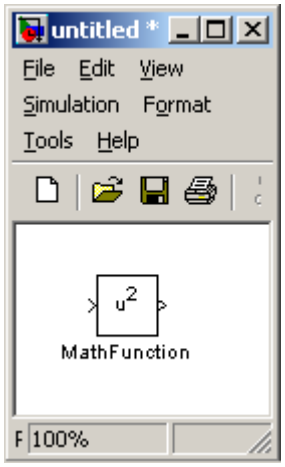
Out1 = (double) ((int) In1%(int) In2)

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

7.4.3.4 Associated EPICS Records

None.

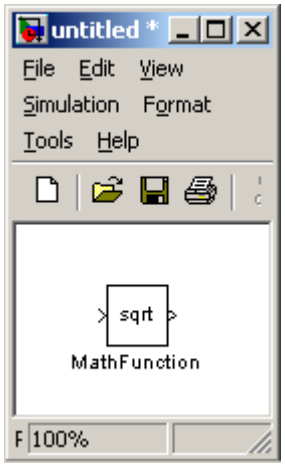
7.4.3.5 Code Examples



The MathFunction module generates the following C code:

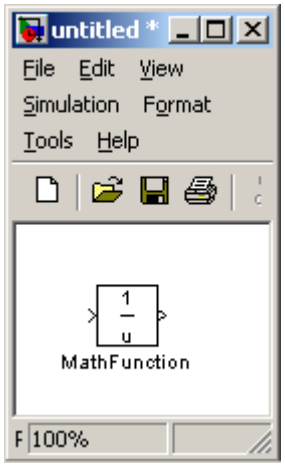
Square:

```
double mathfunction;

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

<Out1> = mathfunction;
```



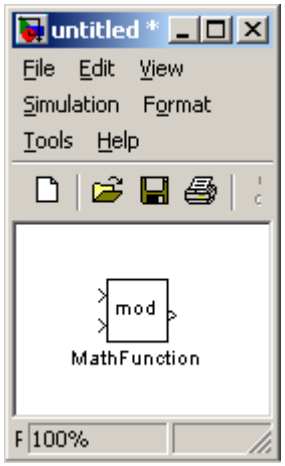
Square root:

```
double mathfunction;

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

<Out1> = mathfunction;
```



Reciprocal:

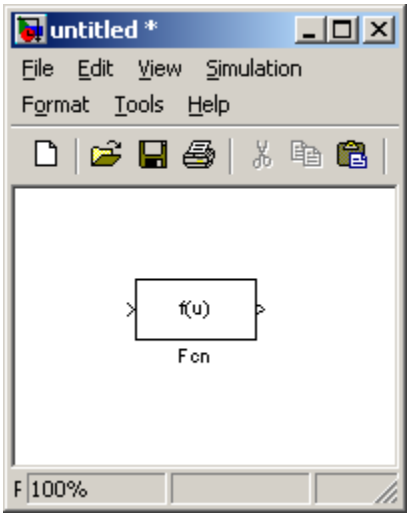
```
double mathfunction;

// 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;
```

7.4.4 In-line (math) function



7.4.4.1 Function

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

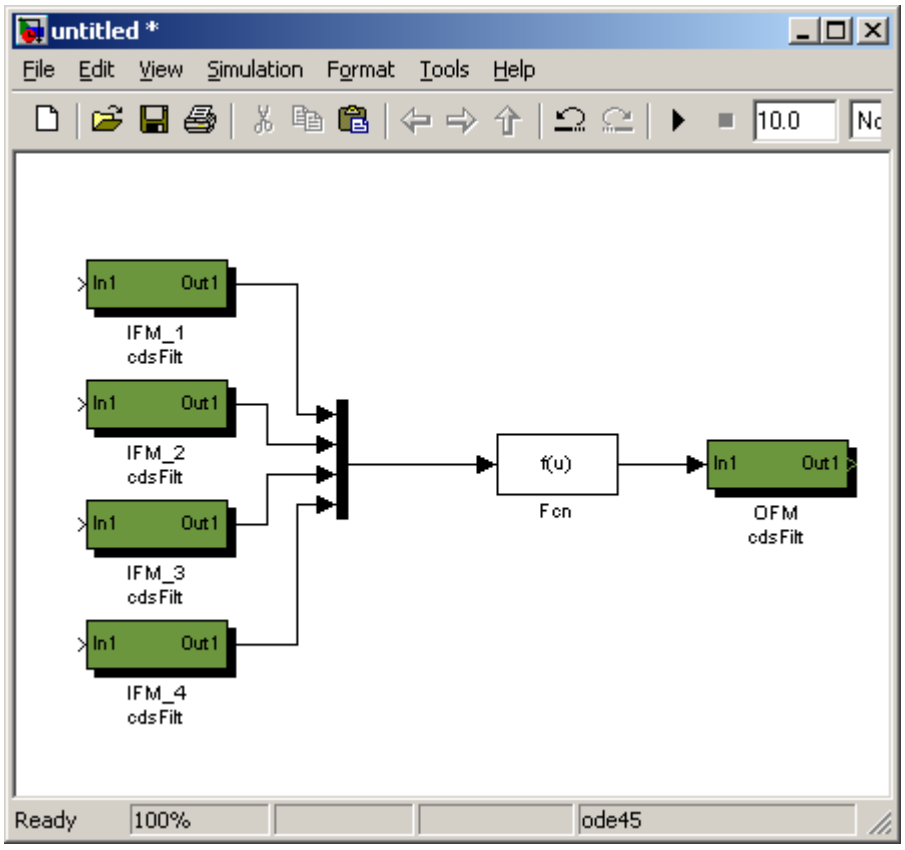
7.4.4.2 Usage

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

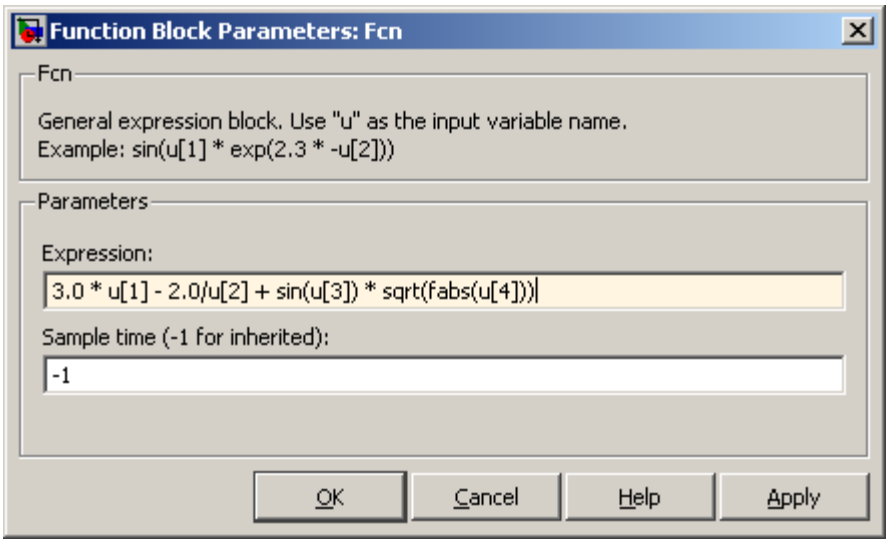
- Polynomials.
- Non-polynomial combinations of variables and constants.
- Sines and cosines.
- Floating-point absolute values.
- log10.
- Square root.
- Combinations of the above.

7.4.4.3 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):

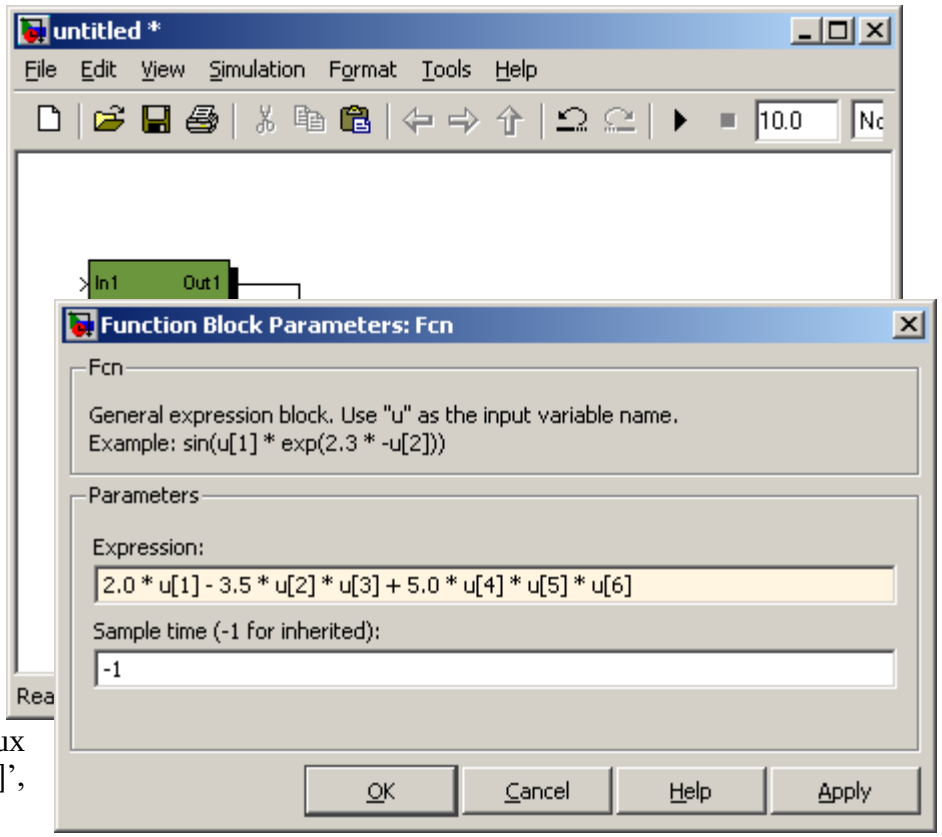


Once the function has been defined, click on OK and the function will be incorporated into the model. Please note that it is up to the user to ensure the validity of entered functions and values, e.g., only positive values for logarithms, no negative values for square roots, no divisions by zero, etc. 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'.

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).

7.4.4.4

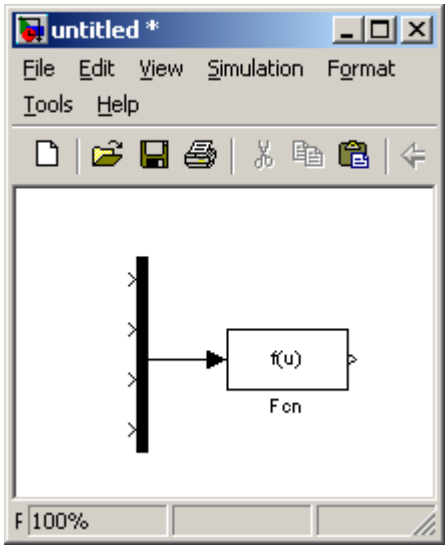
Associated EPICS Records

None.

7.4.4.5 Code Examples

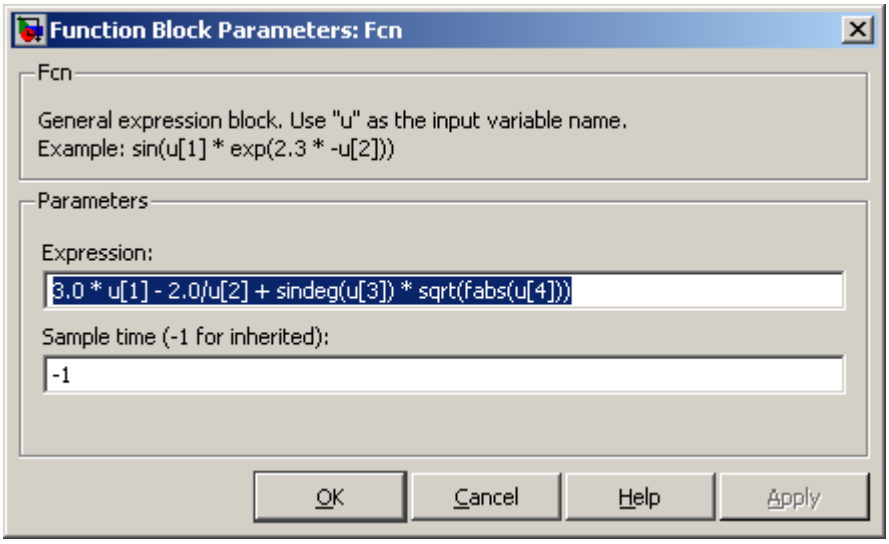
The in-line (math) function generates the following C code:

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

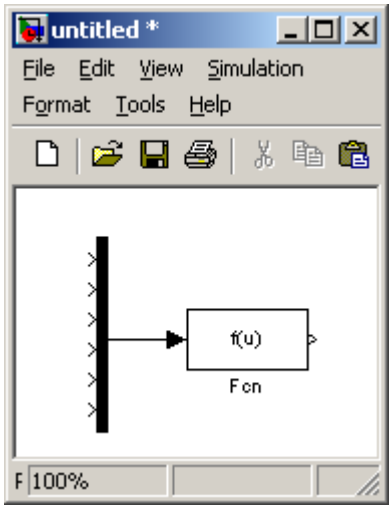



```
double fcn;
double conv = 3.141592654/180.0;
double lcos1, lsin1;

double mux[4];

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

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

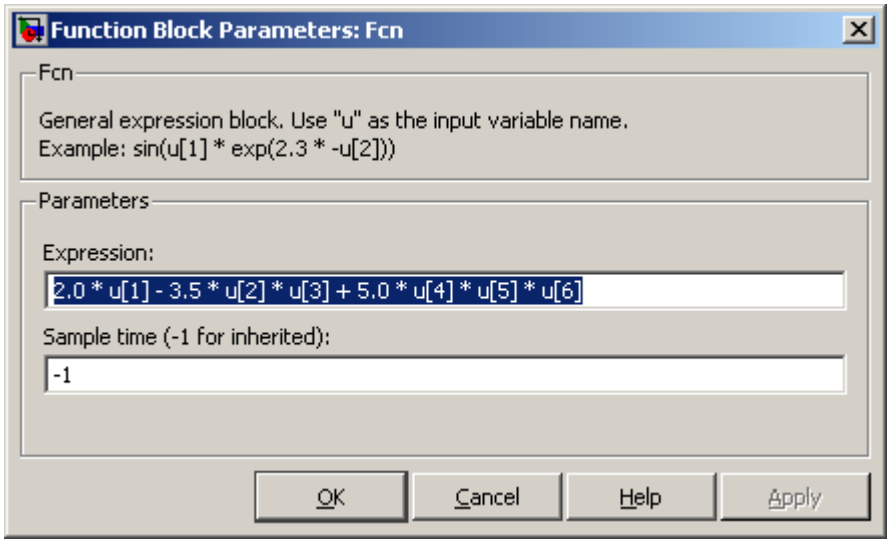
<Out1> = fcn;


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:
8.0 * u[1] - 2.0/u[2] + sindeg(u[3]) * sqrt(fabs(u[4]))
Sample time (-1 for inherited):
-1
OK
Cancel
Help
Apply

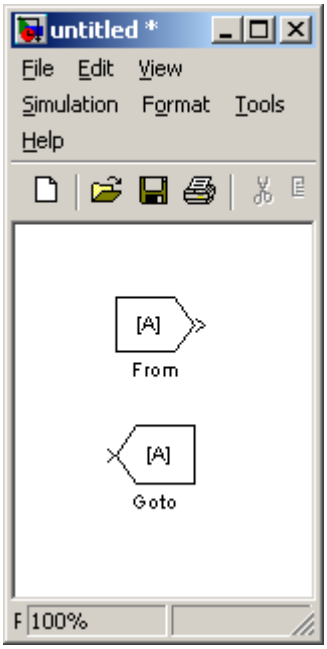
(This example is identical to the second example in section 7.3.4.3.)




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

<Out1> = fcn;
```



7.4.5 From/Goto



7.4.5.1 Function

Connect signals between components in a model without the use of lines ie help provide a cleaner diagram.

7.4.5.2 Usage

GOTO part must be defined with a unique name. To connect that signal to a FROM, the name of the GOTO must be provided.

7.4.5.3 Operation

Only used by RCG for signal routing in compilation.

7.4.6 Bus Creator / Bus Selector

7.4.6.1 Function

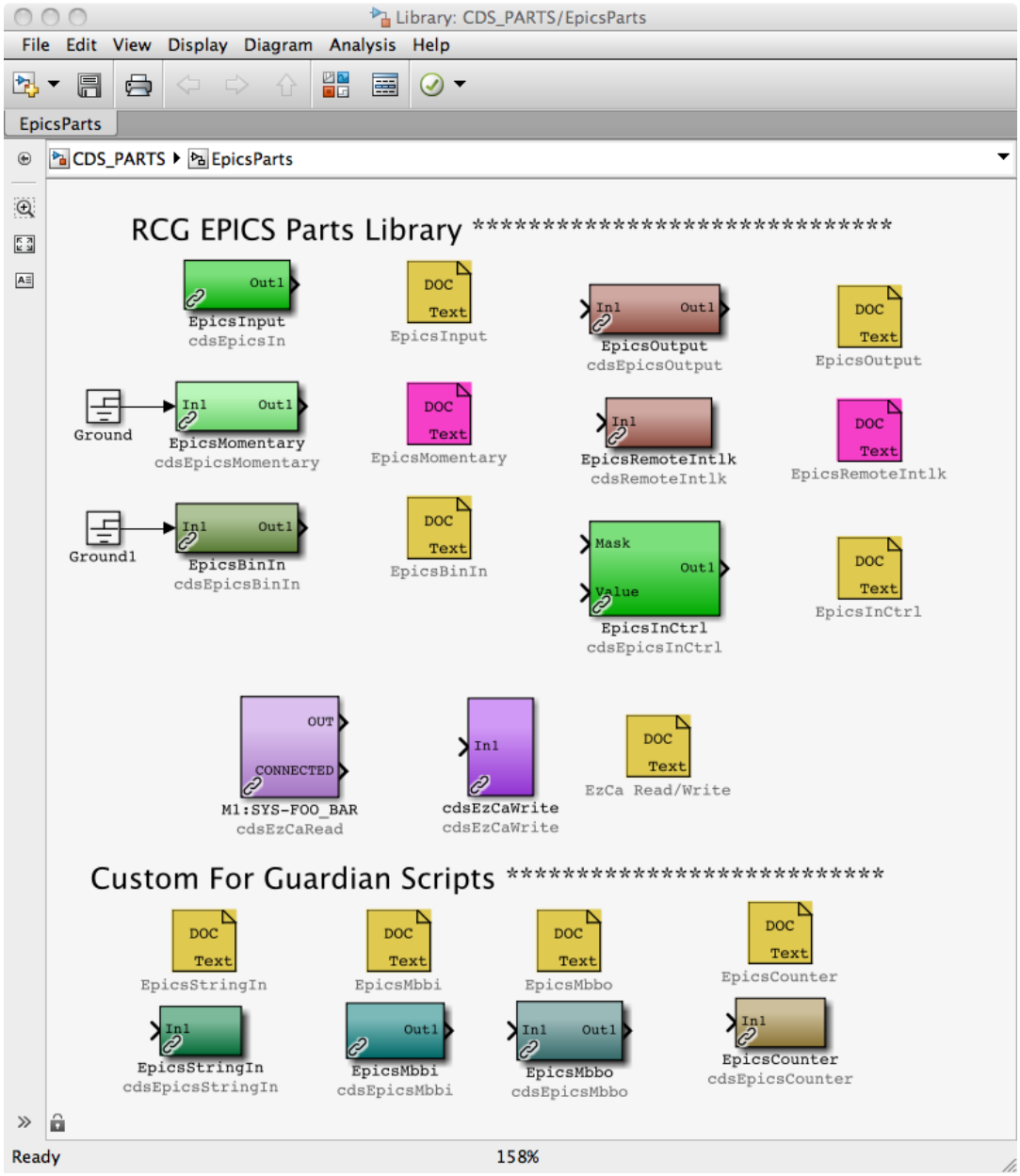
Support for the Matlab standard bus creator/selector parts has been added in version 2.x of the RCG. It's primary function is to allow signal connections between various model components with fewer line drawings required, which in turn, provides for a cleaner model appearance.

7.4.6.2 Usage

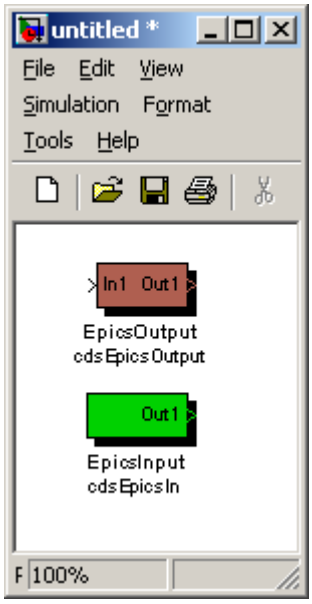
1) Place the part in the model.
2) Double click the part, which brings up a dialog box.
3) Enter the number of inputs or outputs desired.
4) Connect inputs/outputs to other parts within the model.

7.5 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.



7.5.1 cdsEpicsOutput/cdsEpicsIn



7.5.1.1 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.*

7.5.1.2 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.

7.5.1.3 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.

7.5.1.4 Associated EPICS Records

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

7.5.1.5 Setting EPICS Database Fields

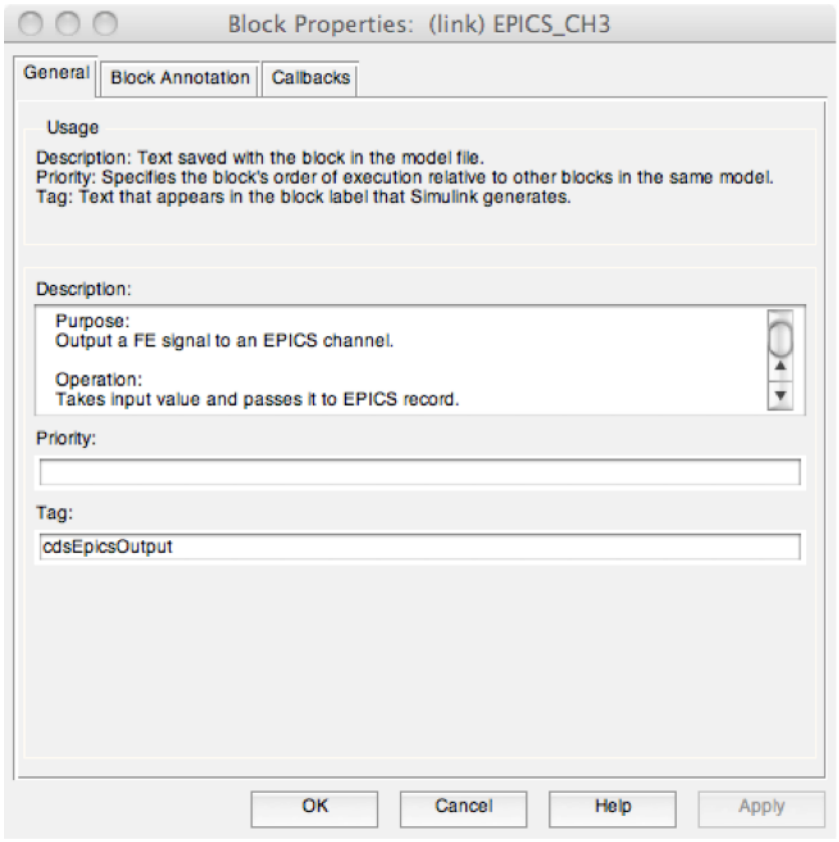
EPICS database records have a number of parameters, or fields, which may be set as part of the database record definition file. For each model compiled with the RCG, a corresponding EPICS database file is created for runtime support.

By default, the RCG only sets the precision of EPICS input and output records in the database file (PREC=3), which provides 3 decimal places of precision when viewed on an MEDM screen.

The RCG does allow users to define parameter fields for the EPICS Input and Output part types within the user model, as described below. A complete list of parameters supported by EPICS AO and AI record types can be found in the EPICS user guide online.

To define these EPICS fields:

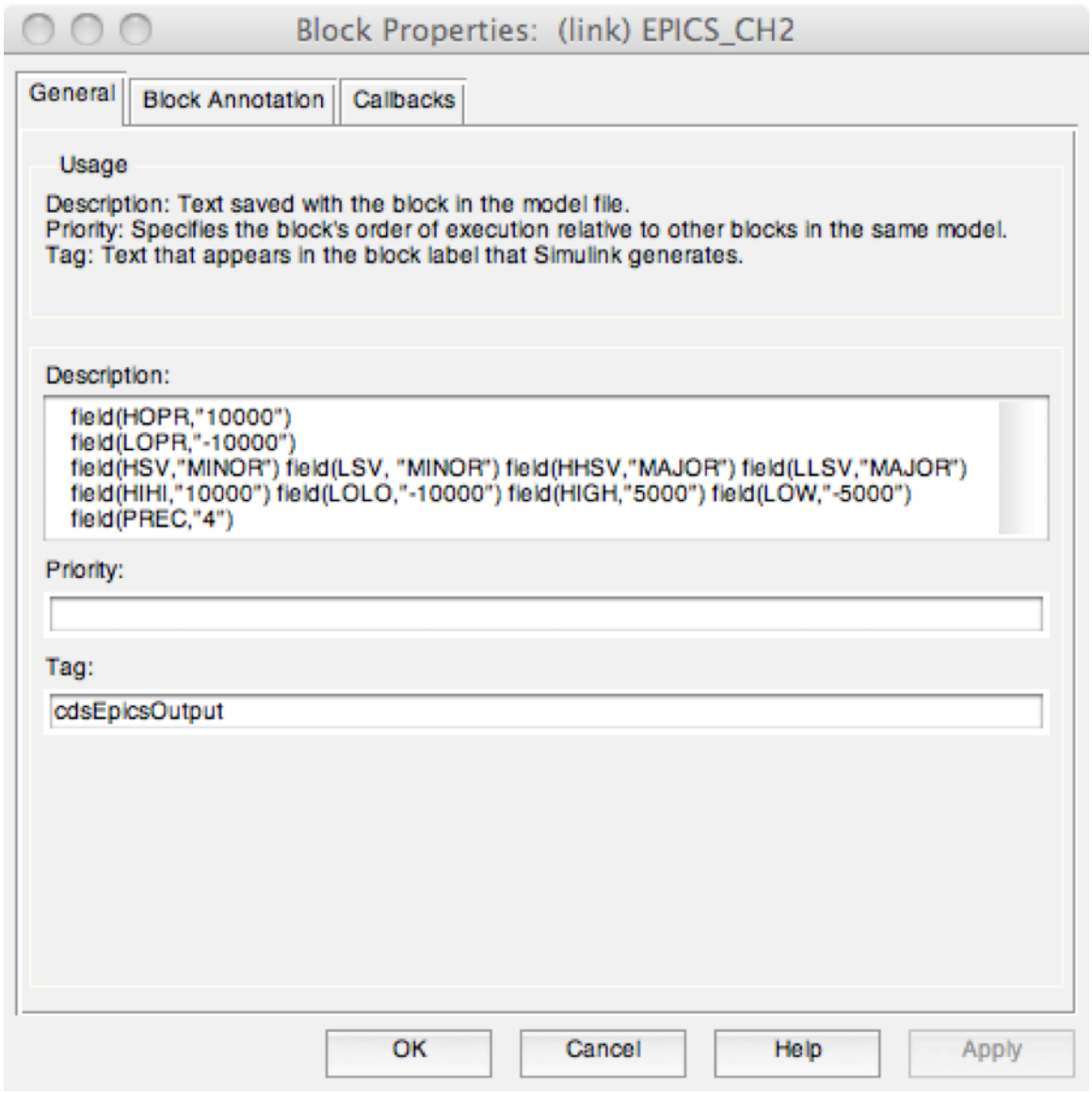
- Place an EPICS Input or Output part into the model and provide a name for the part.
- Open the block properties window for the part. By default, the Description field provides some basic info on the part (Figure 1 below).



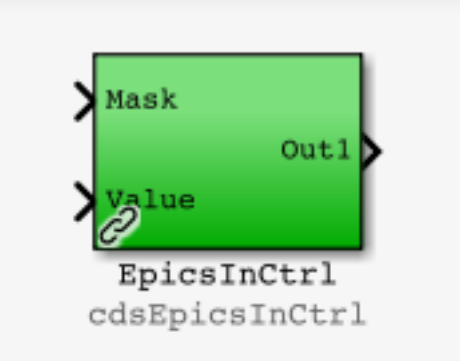
- Delete the provided Description information. (While RCG will ignore this default information, it is probably best to delete it for ease of reading later).
- Add EPICS database parameter information, as shown in Figure 2 below, in the Description area.
    - Each entry must be of the form 'field(PARAM,"VALUE")', where:
        - PARAM = The EPICS parameter definition, such as PREC, HIGH, LOW, etc. The most commonly used are:
            - PREC (Precision), number of decimal places returned to MEDM screens for viewing. Note that this does not affect the calculation precision ie all EPICS values are treated as doubles in the runtime code.
            - HOPR (High Operating Range)
            - LOPR (Low Operating Range)
            - Alarm Severities: HHSV, HSV,LSV,LLSV.
            - Alarm Setpoints: HIHI, HIGH, LOW, LOLO
        - VALUE = Desired default setting, which must be in quotes.
            - Alarm Severities are limited to the following:
                - MAJOR
                - MINOR
                - INVALID
                - NO_ALARM (Default, if not specified)
            - Other entries listed above are all taken as floating point numbers.

- Field definition entries may be separated by white space or new lines, or both, as shown in the example below.

WARNING: Presently, the RCG does not perform any checking of the validity of user definitions provided with the field entries. As long as the entry is of the right form, the RCG will add it to the database definition file. Therefore, it is the user responsibility to ensure entries are correct. Entry error checking is presently being worked for RCG release V2.7 and later.



7.5.3 EpicsInCtrl



7.5.3.1 Function

The purpose of this block is to allow local control, ie from within the real-time code, of EPICS input variables.

7.5.3.2 Usage

Beyond the single output connection of the standard EPICS Input part, this part has two inputs added:

- Mask: This input controls what the input to this part will be:
  - From the associated EPICS record (remote control), which is the standard input for the EPICS input part.
  - Or from the real-time code (local control) via the Value input.

7.5.3.3 Operation

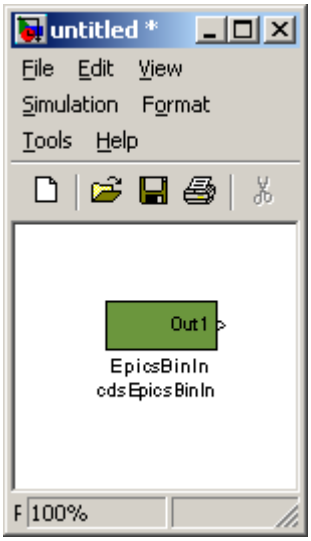
Operation of this part is primarily based on the Mask input signal.

- If Mask input = 0, this part will receive its input from the associated EPICS record ie remote control.
- If Mask=1, this indicates local control. The output of this block will be the Value input on the second input connection. This value will also be sent to EPICS, causing the associated EPICS record to take on this value. In this manner, when the mask is set back to zero and inputs are coming from EPICS, the value set locally will remain the same. NOTE: When the Mask input is set to 1 and later returned to 0, there will be a one second delay before remote control is re-enabled. This is done to prevent a race condition between EPICS and local control ie ensure EPICS has the updated information prior to switching back to remote control.

7.5.3.4 Associated EPICS Records

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

7.5.5 cdsEpicsBinIn



7.5.5.1 Function

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

7.5.5.2 Usage

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

7.5.5.3 Operation

Out1 = EPICS value placed in shared memory.

7.5.5.4 Associated EPICS Records

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

7.5.6 cdsRemoteIntlk

7.5.6.1 Function

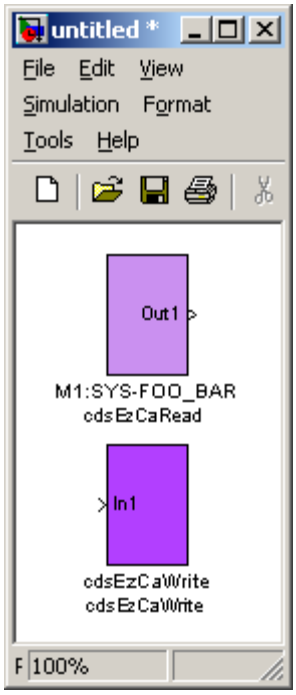
7.5.6.2 Usage

7.5.6.3 Operation

7.5.6.4 Associated EPICS Records

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

7.5.7 cdsEzCaRead/cdsEzCaWrite



7.5.7.1 Function

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

7.5.7.2 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-.

7.5.7.3 Operation

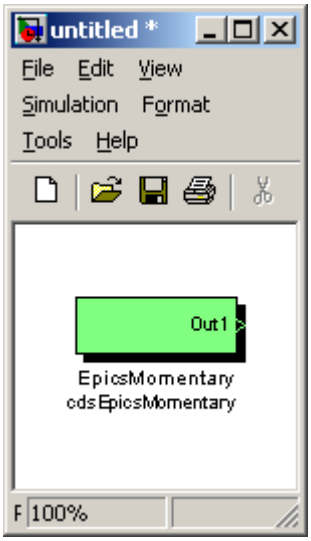
The EPICS sequencer which supports the real-time code will have EzCaRead/EzCaWrite commands added to obtain/set the desired values via the Ethernet. Values are passed out of/into the real-time code via shared memory.

7.5.7.4 Associated EPICS Records

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

7.5.7.5 Code Examples

7.5.8 EPICS Momentary



7.5.8.1 Function

The cdsEpicsMomentary module is used to flip one bi

7.5.8.2 Usage

…

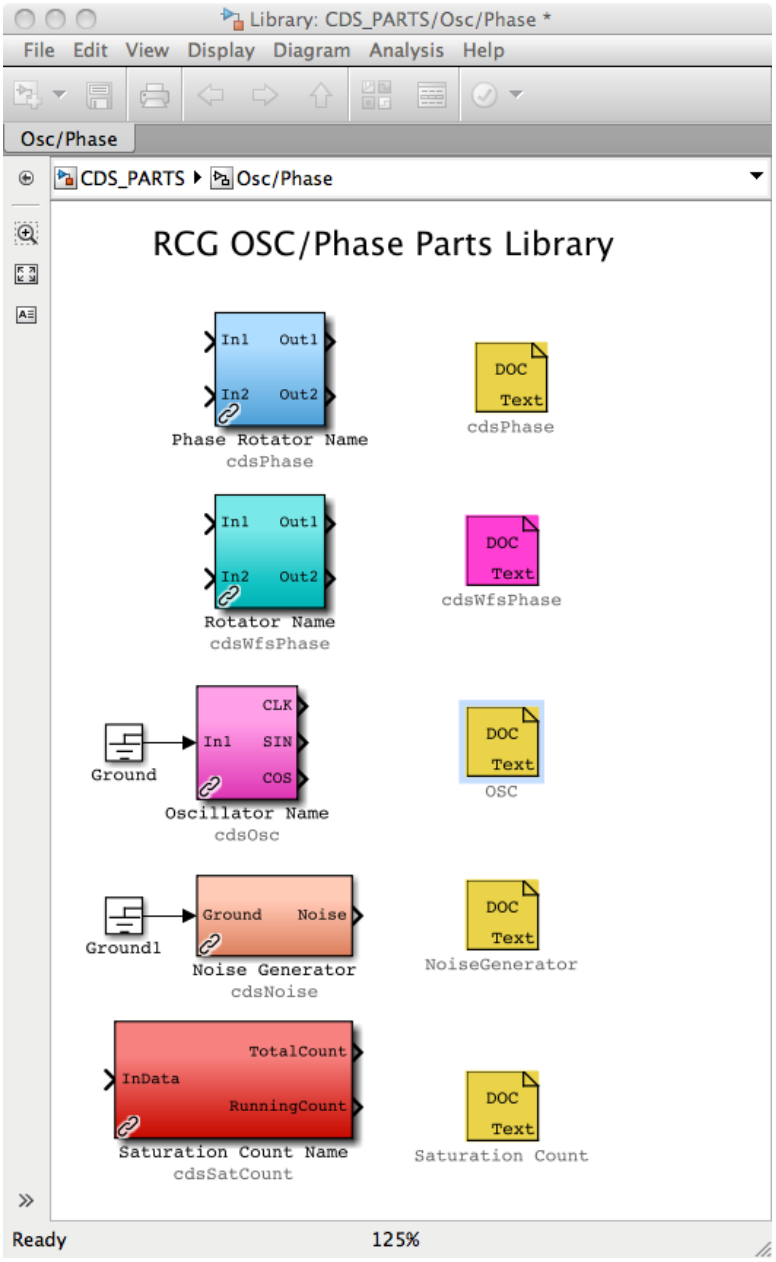
7.5.8.3 Operation

7.5.8.4 Associated EPICS Records

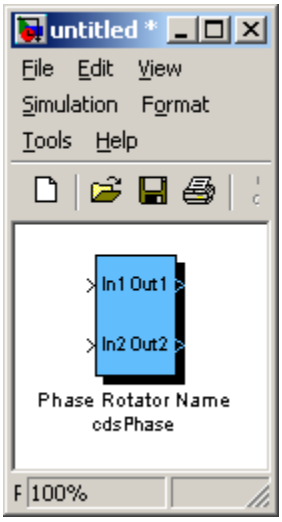
A momentary ‘ai’ EPICS record switch will be produced using the name assigned to this block.

7.6 Osc/Phase

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



7.6.1 cdsPhase



7.6.1.1 Function

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

7.6.1.2 Usage

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

7.6.1.3 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:

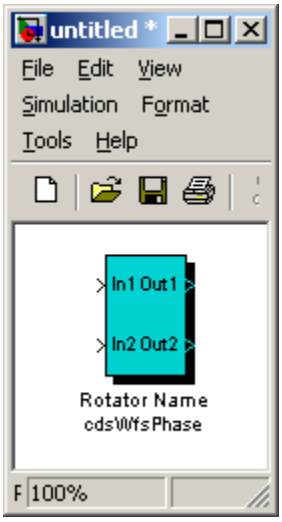
Out1 = In1 * cosPhase + In2 * sinPhase

Out2 = In2 * cosPhase – In1 * sinPhase

7.6.1.4 Associated EPICS Records

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

7.6.2 cdsWfsPhase



7.6.2.1 Function

7.6.2.2 Usage

7.6.2.3 Operation

7.6.2.4 Associated EPICS Records

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

7.6.3 cdsOsc

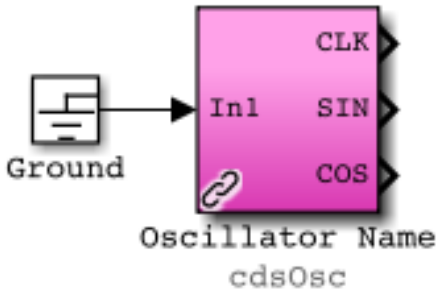
7.6.3.1 Function

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

7.6.3.2 Usage

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

NOTE: This part still requires a GROUND at its input to compile properly (bug yet to be fixed).



7.6.3.3 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' output 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.

When changing a gain, if the TRAMP channel is set to 0 (or below), it will instantly change the gain. A positive TRAMP value will cause the gain to perform a spline ramp ot the new gain over the a number of seconds equal to the value.

When changing frequency, if the TRAMP channel is set to 0 (or below), it will change frequency at the next GPS second (as clocked by the front end). It will have an initial phase of 0. If the TRAMP channel is positive, it will immediately start ramping to the new frequency over a number of seconds equal to the value. It will have a phase such that at the next GPS second after it finishes ramping it will have a phase of 0.

7.6.3.4 Associated EPICS Records

Four EPICS records are produced for user entries:

_FREQ: Desired frequency in Hz

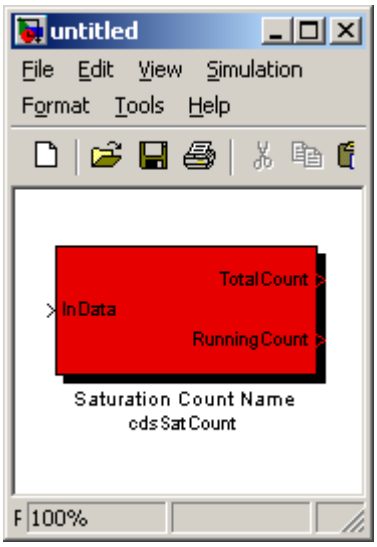
_CLKGAIN: CLK gain setting

_SINGAIN: SIN gain setting

_COSGAIN: COS gain setting

_TRAMP: Time to do gain and frequency ramping, in seconds.

7.6.4 cdsSatCount



7.6.4.1 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.

7.6.4.2 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.

7.6.4.3 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.

7.6.4.4 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).

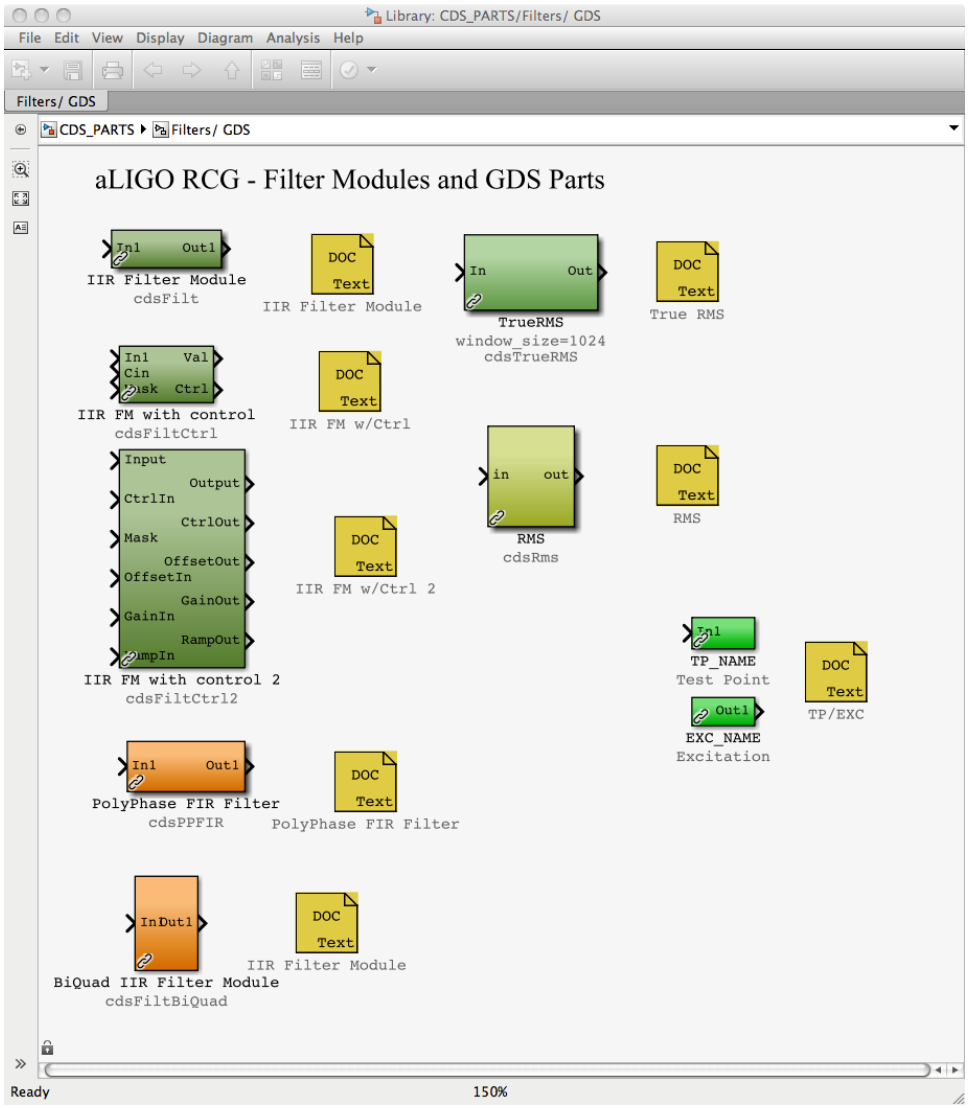
_TRIGGER: The TotalCount and RunningCount counters (and outputs) will increment as long as the absolute value of the channel (input) datum is greater than or equal to the TRIGGER threshold value (initial default TRIGGER value is equal to zero)

7.6.5 cdsNoise

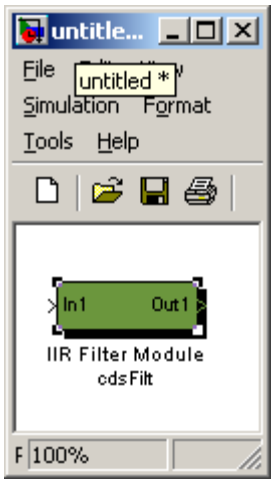
7.8 Filters

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.



7.8.1 CDS Standard IIR Filter Module



7.8.1.1 Function

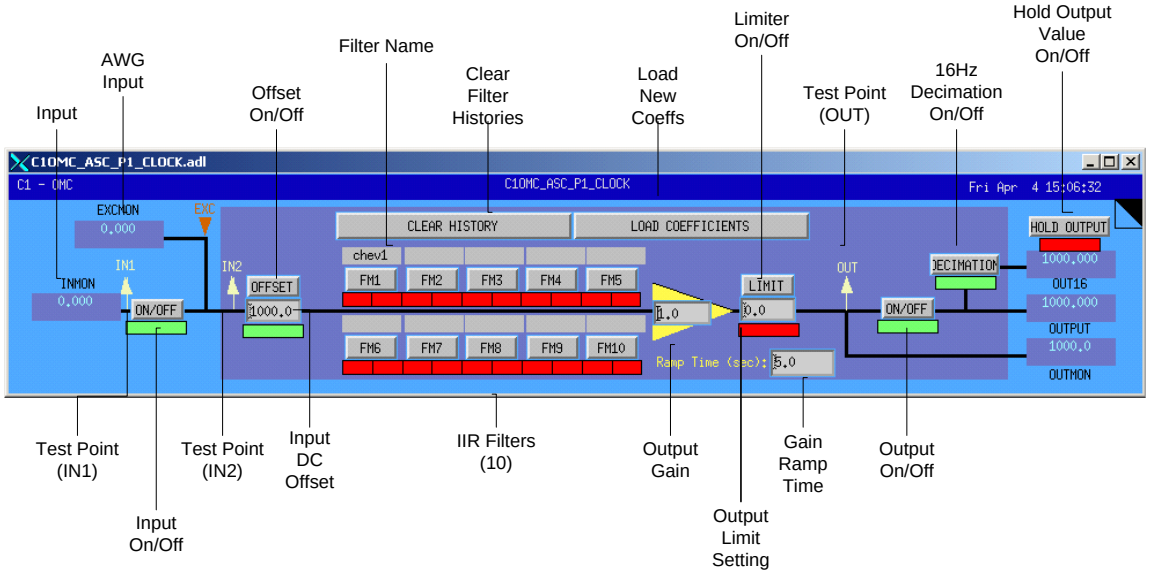
All CDS FE processors use digital Infinite Impulse Response (IIR) filters to perform a majority of their signal conditioning and control algorithm tasks. In order to facilitate their incorporation into FE software and to provide a standard set of DAQ and diagnostic capabilities, the Standard Filter Module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

7.8.1.2 Usage

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

7.8.1.3 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).



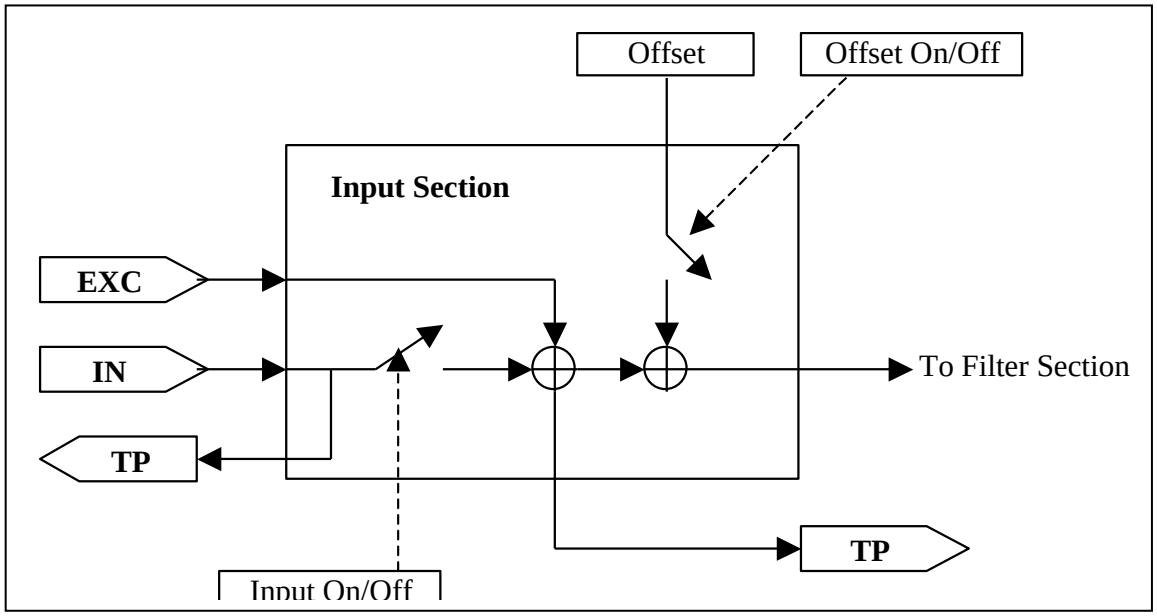
7.8.1.3.1 Input Section

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).

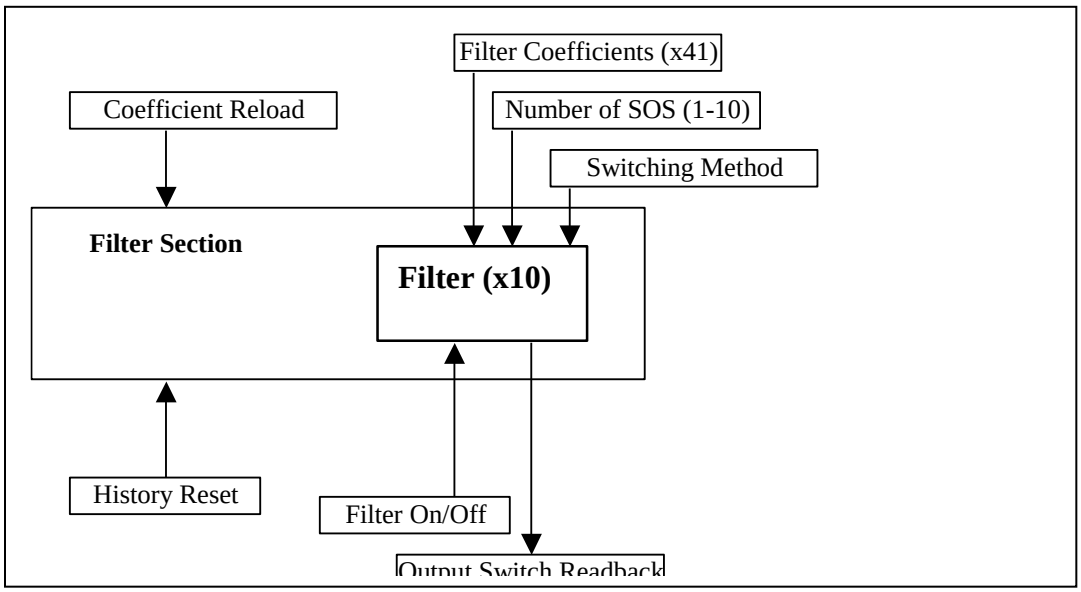
To this resulting signal, a DC offset may be added (Input DC Offset) and this offset may be turned on/off via the Offset on/off switch. The sum of the input, AWG and offset signal is then fed to the IIR filtering section.



7.8.1.3.2 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.

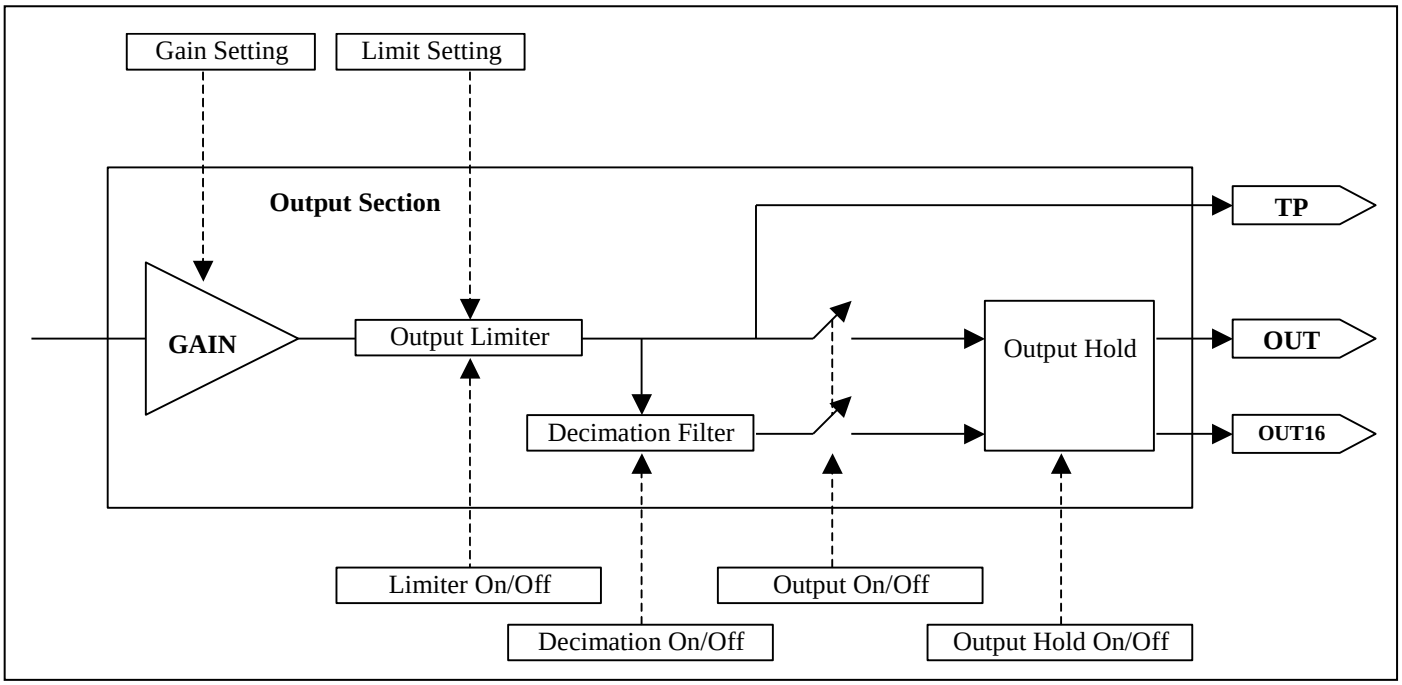
	• 2 – Input Crossing: The output will switch when the filter input and output are within a given value of each other. This value is contained in the RAMP field. • 3 – Zero Crossing: The output will switch when the filter input crosses zero.
Number of SOS	This field contains the number of Second Order 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.

7.8.1.3.3 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.



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)

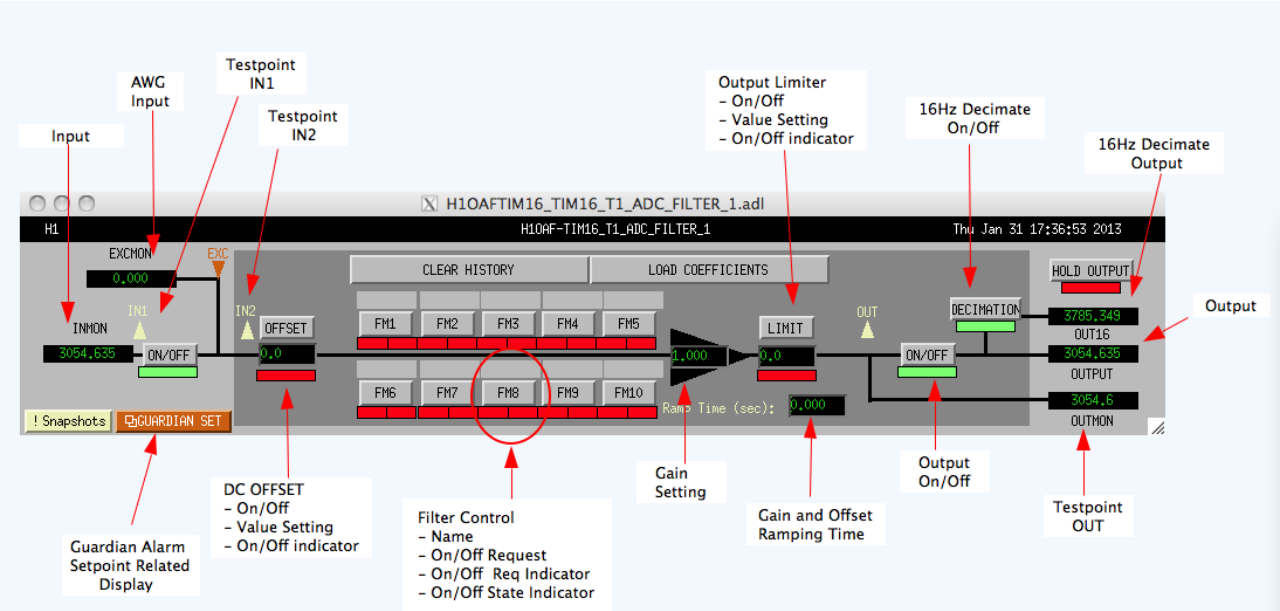
_SW1S = Saved filter switch selections, lower 16 bits (RO)

_SW2S = Saved filter switch selections, upper 16 bits (RO)

_Name00 thru _Name09 = Individual filter names, as defined in the coefficient file (RO)

7.8.1.4 Auto-Generated MEDM Screens

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.

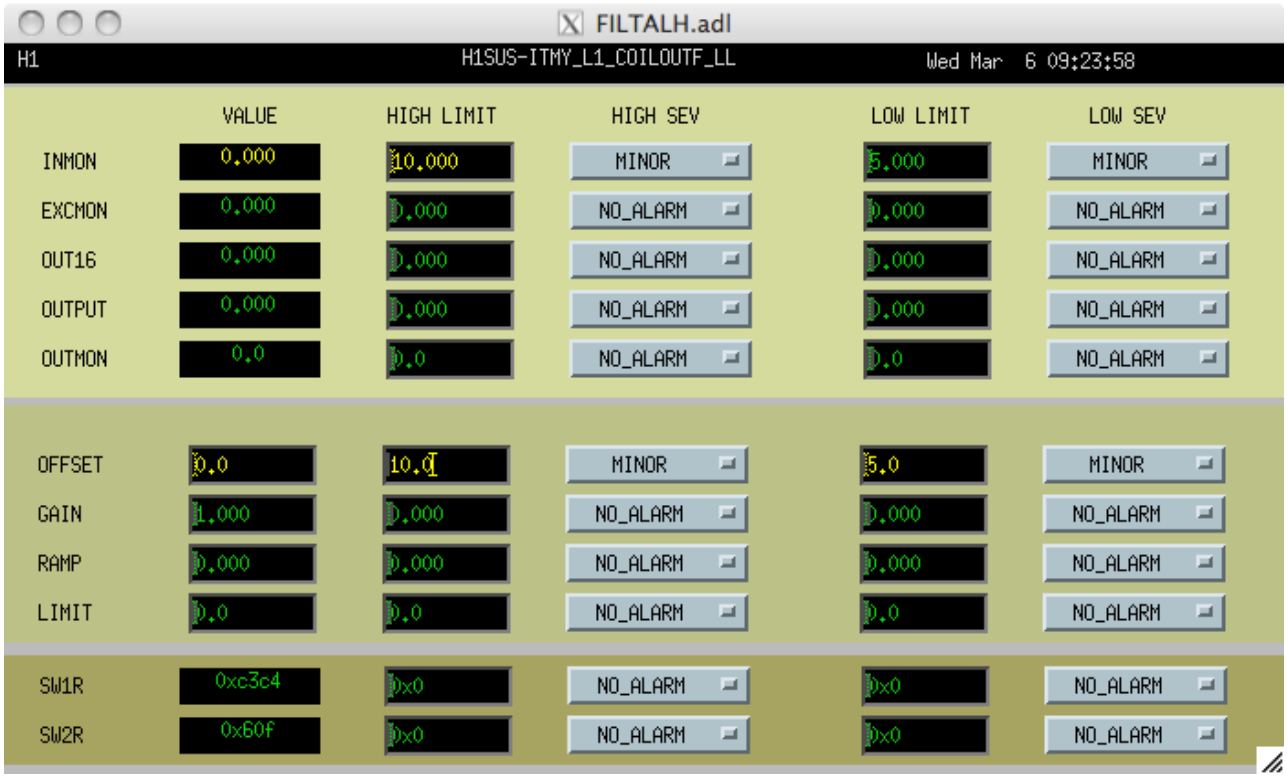


This screen contains the following EPICS I/O:

- 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.

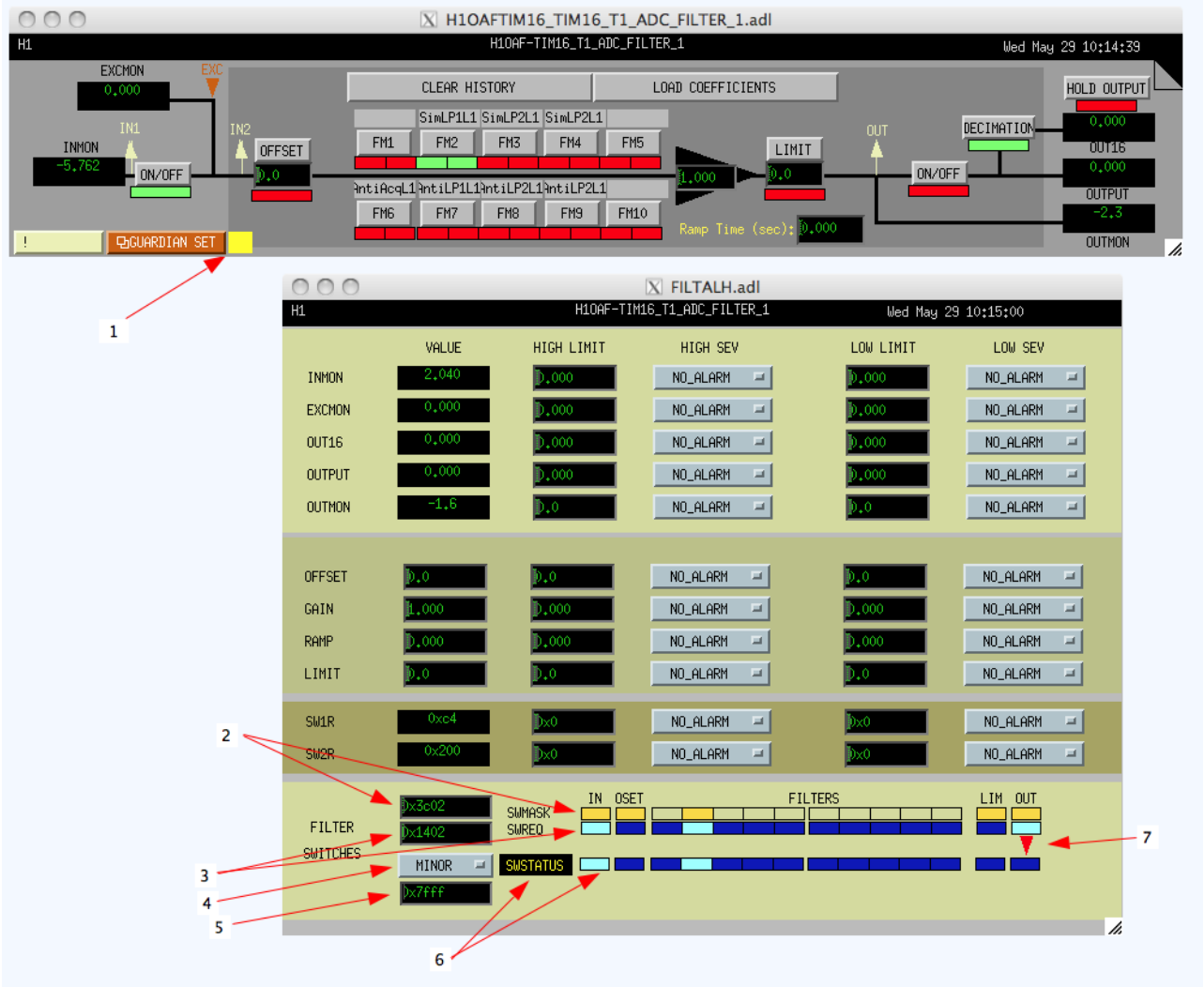


In RCG V2.7, this screen has been modified with a lower section of new indicators and settings, as shown in the following figure:

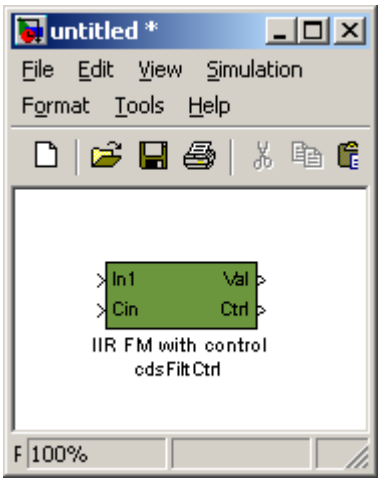
1. SWMASK Setting: A text entry block is provided, along with indicators of the bit settings that correspond to this setting. If a MASK bit is set, the indicator will be YELLOW. This indicates which bits the Guardian script requires to be as set by the SWREQ word. In this example, MASK indicates that Guardian requires Input, Offset, Filter 2, Limit and Output switches to be set as indicated by the SWREQ setting. All other switch settings are 'Don't Care', ie may be either ON or OFF.
2. SWREQ Setting: Text entry point for setting the Guardian requested switch states and bit pattern indicators. If a switch is required to be ON, then indicator is LIGHT BLUE. If required OFF, indicator is DARK BLUE. In this example, Guardian requires:

    - Input Switch ON
    - Offset Switch OFF
    - Filter Switch 2 ON
    - Limit Switch OFF
    - Output Switch ON

3. Alarm Severity Setting: In order for Guardian to be notified of a mismatch between required and actual switch settings, the ALARM SEVERITY must be set.
4. Alarm Limit: This should always be set to 0x7fff (32767), as bit 15 set is the alarm bit, as previously described.
5. SWSTAT: Bit indicators of the present SFM switch settings and alarm indication (SWSTATUS in YELLOW/RED if alarm state).
6. Error indicators (RED arrows), pointing out which switches are not set as required by Guardian. In this example, Output switch is set to OFF, whereas Guardian requirement is that this switch be ON.



7.8.2 IIR Filter Module with Control



7.8.2.1 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.

7.8.2.2 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.

7.8.2.3 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.

7.8.2.4 Associated EPICS Records

Same as cdsFilt module.

7.8.2.5 Auto-Generated MEDM Screens

Same as those provided for cdsFilt part.

7.8.3 IIR Filter Module with Control 2

7.8.3.1 Function

This part is similar in function to the IIR Filter Module with Control, described in the previous section. However, it has additional inputs/outputs defined to control more settings from within a user control model. The three new inputs allow for setting of the FMC2 offset, gain and ramp time from within the user control model. The new outputs provide information on the present settings for these three filter module parameters, regardless of whether or not a value is under local or remote control.

7.8.3.2 Detailed Description

FMC2:

INPUTS
In1 = Signal Input
Cin = Control word input (16 bits)
Mask = 16 bit word, defining which switch and other settings are to be controlled locally (within model).
Offset = Local FM offset setting
Gain = Local FM gain setting
Ramp = Local FM Ramp time (seconds)

OUTPUTS
Val = Output signal from filter module
Ctrl = Present FM switch settings.
Offset = Present FM offset value
Gain = Present FM gain value
Ramp = Present FM Ramp time (seconds)

The bit patterns for the Cin and Mask inputs and Ctrl output are also changed for the FMC2 part. These are all now 16bit words, as defined in the following table. Note that "Local Control" is defined as setpoint control from within the real-time code (user model) and "Remote Control" is defined as control from outside of the real-time code via EPICS Channel Access (ECA), such as from operator MEDM screens, EPICS scripts, etc.

Table 1: Cin and MASK Input and Ctrl Output Word Bit Definitions

Bit	Cin Setting Request 0 = Off, 1 = On	MASK Local/Remote Control Set 0 = Remote, 1 = Local	Ctrl Switch Setting Readout 0 = Off, 1 = On
0	Filter 1 on/off	Filter 1 L/R control	Filter 1 on/off
1	Filter 2 on/off	Filter 2 L/R control	Filter 2 on/off
2	Filter 3 on/off	Filter 3 L/R control	Filter 3 on/off
3	Filter 4 on/off	Filter 4 L/R control	Filter 4 on/off
4	Filter 5 on/off	Filter 5 L/R control	Filter 5 on/off
5	Filter 6 on/off	Filter 6 L/R control	Filter 6 on/off
6	Filter 7 on/off	Filter 7 L/R control	Filter 7 on/off

7	Filter 8 on/off	Filter 8 L/R control	Filter 8 on/off
8	Filter 9 on/off	Filter 9 L/R control	Filter 9 on/off
9	Filter 10 on/off	Filter 10 L/R control	Filter 10 on/off
10	FM Input Switch on/off	FM Input Switch L/R	FM Input Switch on/off
11	FM Offset Switch on/off	FM Offset Switch L/R	FM Offset Switch on/off
12	FM Output Switch on/off	FM Output Switch L/R	FM Output Switch on/off
13	Not Used	FM Offset Setting L/R	FM Offset Setting L/R
14	Not Used	FM Gain Setting L/R	FM Gain Setting L/R
15	Not Used	FM Ramp Time L/R	FM Ramp Time L/R

It should be noted that the Cin input only requires, and RCG code only recognizes, bits 0 through 12. Bits 13 through 15 appear as part of the Ctrl output as a reflection of the upper 3 bits in the MASK input.

7.8.3.3 Usage

Local control of FMC2 settings is enabled/disabled via the MASK input. If the MASK input is zero (0), then all settings are controlled remotely. In this case, the part operates in a manner similar to the standard filter module part, with all settings coming via ECA and EPICS data base records. The values presented at the Cin, Offset, Gain and Ramp inputs are ignored.

Setting a bit to one (1) at the MASK input changes control of the associated parameter to local control. The Cin word is now used to select switch settings, and Gain, Offset and Ramp inputs are used to set those parameters (if associated MASK bit set to one (1)). Rather than being read from EPICS (remote control), the selected parameter settings at the FMC2 input are now sent back to EPICS. Since the EPICS records are updated with the local control settings, switching back to remote control will not change the present settings of the FMC2 ie FMC2 will receive the same settings as last written via local control prior to the switch over.

Whether a parameter is in local or remote control, the Ctrl and Offset, Gain and Ramp outputs always reflect the present FMC2 settings. These outputs are provided to allow user code to determine present state prior to switching to local control and/or verification of settings while in local control.

It should be noted that the lower 10 bits of the Ctrl output reflect the present on/off state of the individual filters, not the requested state. Therefore, depending on filter design, there may be a delay between on/off request at the Cin input and the associated on/off bit setting in the Ctrl output word. For example, if a filter is designed, using foton, to switch only on zero crossing, there may be a delay between switching request and actual filter turn on/off.

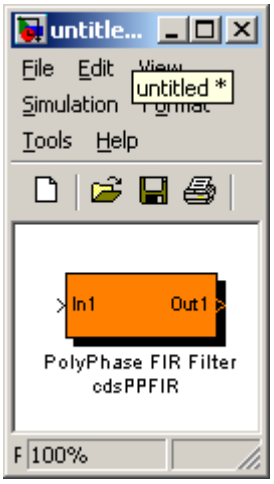
7.8.3.4 EPICS Database Records

Beyond those provided for standard IIR filter modules, an additional record is provided to reflect the MASK input setting. This record is of the form FILTER_MODULE_NAME_MASK.

7.8.3.5 Auto-Generated MEDM Screen

The RCG produces a screen that is similar to that produced for the standard IIR filter part. An added feature is indication of which filter module parameters are presently under control by the RT code model. Near each setting on the screen, an LC (local control) will appear when under RT code control.

7.8.4 PolyPhase FIR Filter



7.8.4.1 Function

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

7.8.4.2 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.

7.8.4.3 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).

7.8.4.4 Associated EPICS Records

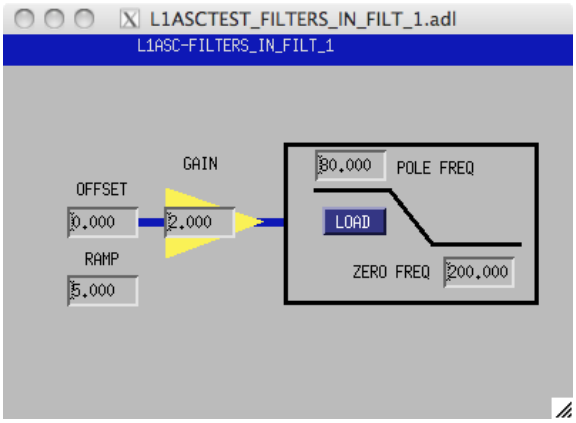
Same as cdsFilt module.

7.8.5 Input Filter (Single Pole / Single Zero (SPSZ) with EPICS control)

This filter module uses pole/zero settings from EPICS.

7.8.5.1 Function

Provides a single pole, single zero filter function, with input settings provided via EPICS.



7.8.5.2 Usage

7.8.5.3 Operation

Given:

- Overall high frequency gain = K
- Z and P are in Hz.
- Fs = code sampling frequency

Calculation:

$a = (1-\pi P/fs)/(1+\pi P/fs)$

$b = (1-\pi Z/fs)/(1+\pi Z/fs)$

val = K * (Input + Offset)

output = val – (b * val_previous) + (a * input_previous)

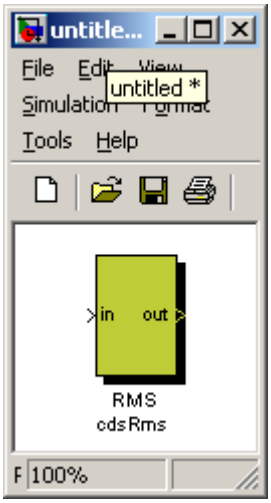
7.8.5.4 Associated EPICS Records

1) _OFFSET : Input offset value
2) _TRAMP: Ramp time, in seconds
3) _K: Gain term
4) _P: Pole term
5) _Z: Zero term
6) _Load: Momentary switch that starts load of new settings over the time specified by TRAMP.

7.8.5.5 Auto-Generated MEDM Screens

As shown in figure above.

7.8.6 RMS Filter



7.8.6.1 Function

This block computes the RMS value of the input signal.

7.8.6.2 Usage

This module is used to calculate an RMS value.

7.8.6.3 Operation

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

7.8.6.4 Associated EPICS Records

None.

7.8.7 True RMS Filter

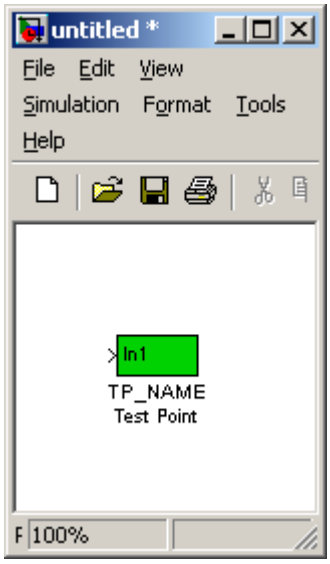
7.8.7.1 Function

This block computes the RMS value of the input signal. It takes the root mean square value of a number of samples equal to the window_size parameter.

7.8.7.2 Usage

After placing the part in the user model, adjust the window_size parameter in the block properties description field. The window-size = number of code cycles over which to calculate the RMS value.

7.8.8 Test Point



7.8.8.1 Function

The test point part allows the definition of a GDS test point anywhere in the model without having to use a “Filter Module” part.

7.8.8.2 Usage

The desired test point signal is connected to the part input and given an appropriate signal name.

7.8.8.3 Operation

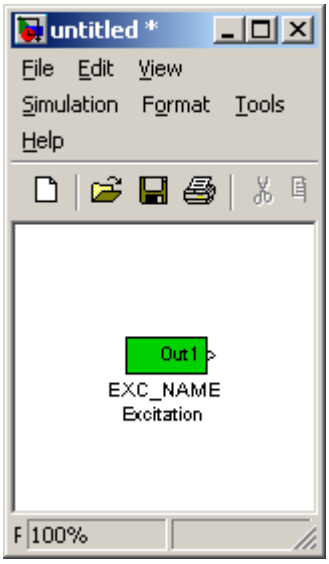
The test point variable will be set equal to the input variable at the full rate of the compiled code. Upon request, this value will become available to the real-time data acquisition software for transmission to the DAQ system.

Note: These signals are also available to be assigned as DAQ channels at user defined rates.

7.8.8.4 Associated EPICS Records

None.

7.8.9 Excitation



7.8.9.1 Function

Provide an input from the GDS arbitrary waveform generator at any point within a user model without having to use a filter module.

7.8.9.2 Usage

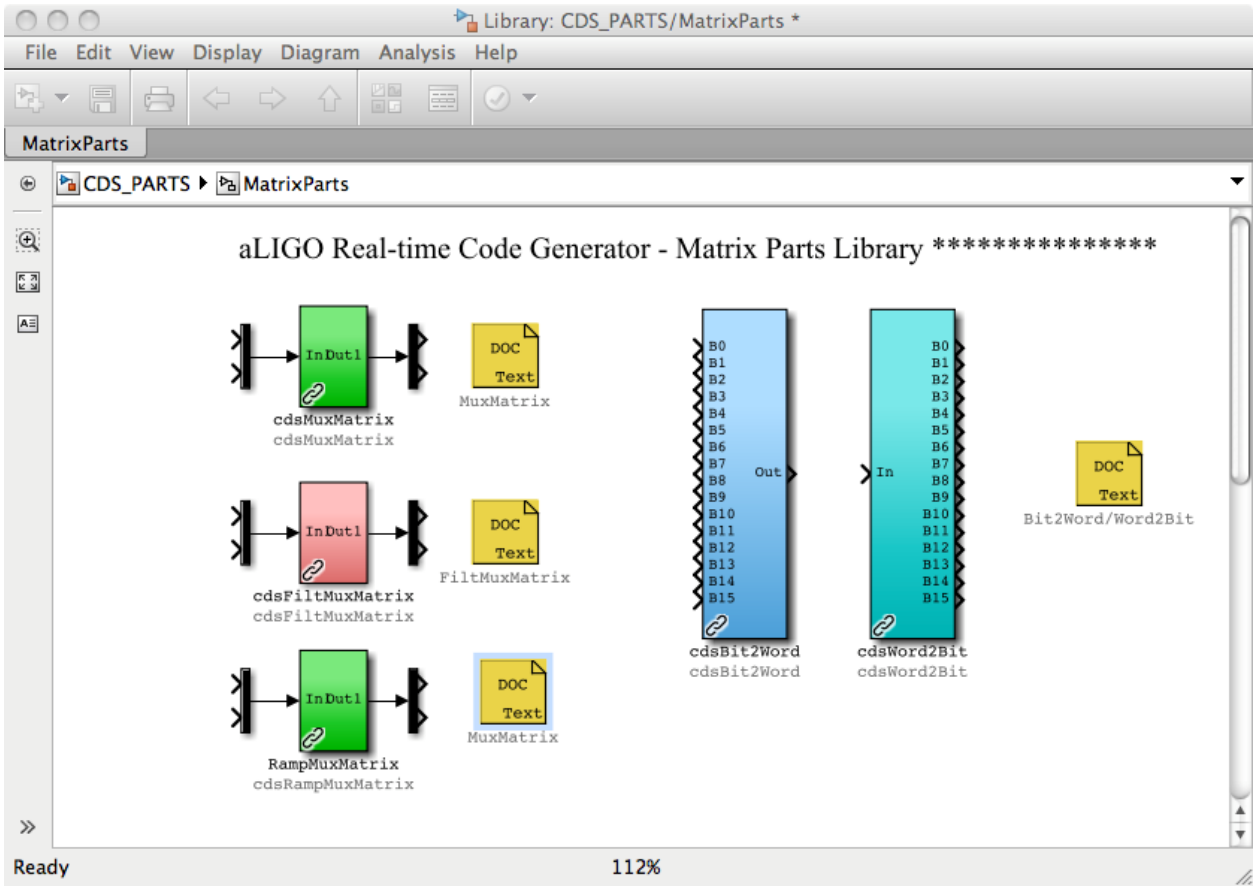
Connect output to any model part with a signal input.

7.8.9.3 Operation

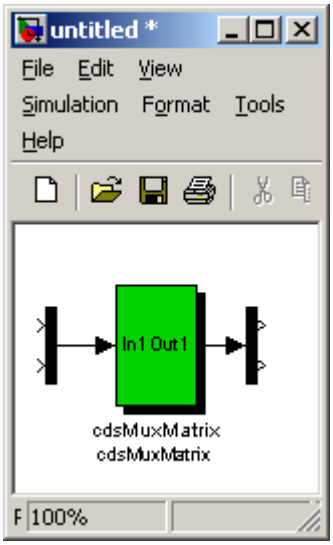
Upon selection via one of the GDS tools, the real-time DAQ process will inject a signal from the arbitrary waveform generator into this variable. If not selected, the output is always zero (0.0).

7.9 Matrix Parts

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



7.9.2 cdsMuxMatrix



7.9.2.1 Function

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

7.9.2.2 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.

7.9.2.3 Operation

Basic code function is:

Output[1] =

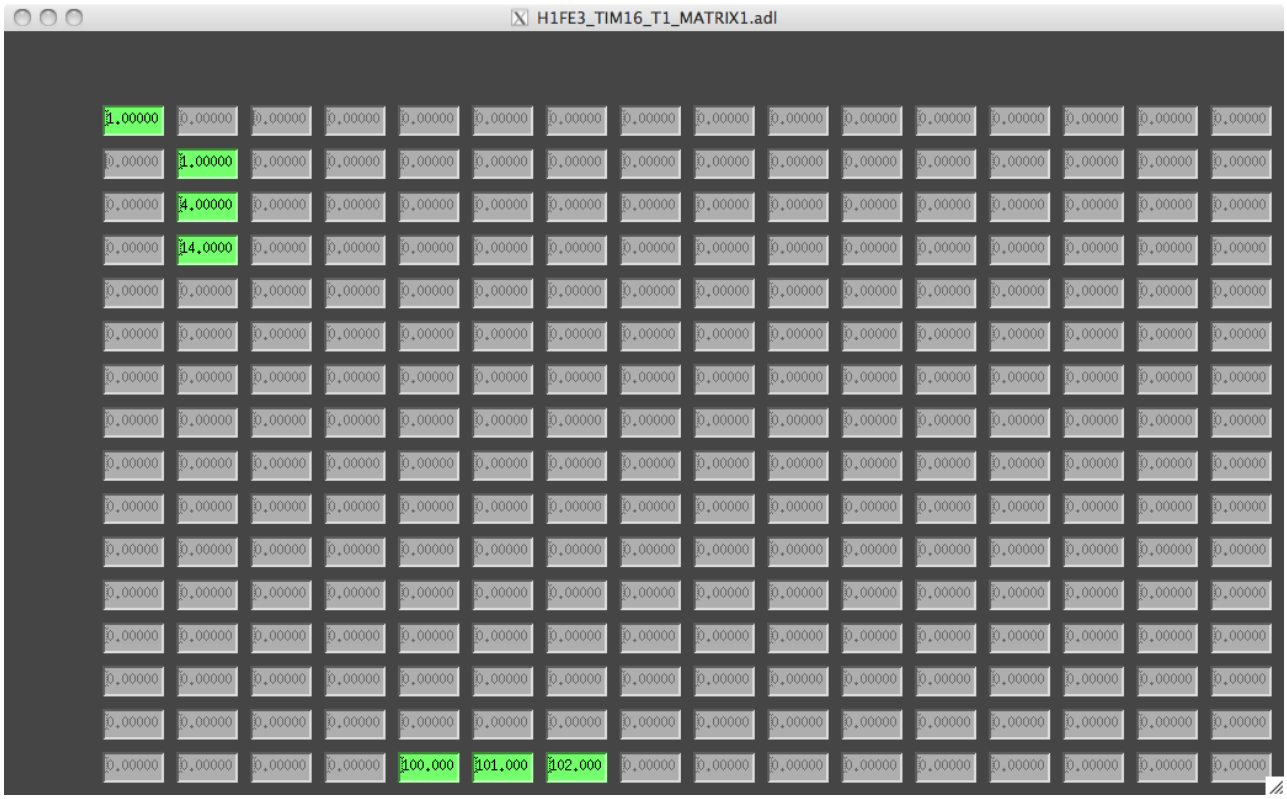
Input[1] * Matrix_11 + Input[2] * Matrix_12 + Input[n] * Matrix_1n, where Matrix_xx is an EPICS entry field.

7.9.2.4 Associated EPICS Records

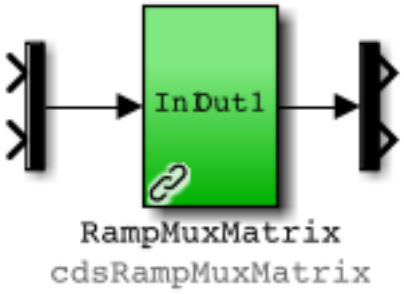
The RCG will produce an A x B matrix of EPICS records for use as input variables, where B is the number of inputs and A is the number of outputs. The EPICS record names will be in the form of PARTNAME_AB, starting at PARTNAME_11.

7.9.2.5 Auto-Generated MEDM Screen

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 gray. Any other value results in a green background.


H1FE3_TIM16_T1_MATRIX1.adl
1.00000
1.00000
4.00000
14.0000
100.000
101.000
102.000

7.9.4 cdsRampMuxMatrix

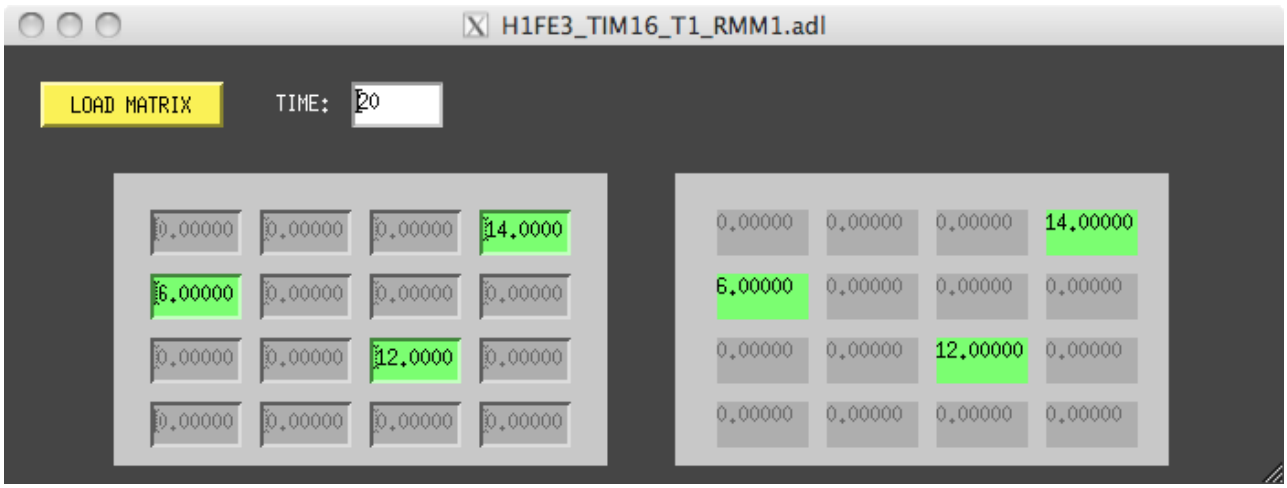


A new matrix part has been added which provides a smooth spline ramping over a user-defined time to load new settings. Operation is perhaps best explained using the example MEDM screens that follow. For this example, a 4x4 matrix has been defined in the model. The RCG produces screens similar to these during compile/install.

As seen in these MEDM screens, there are several new features:

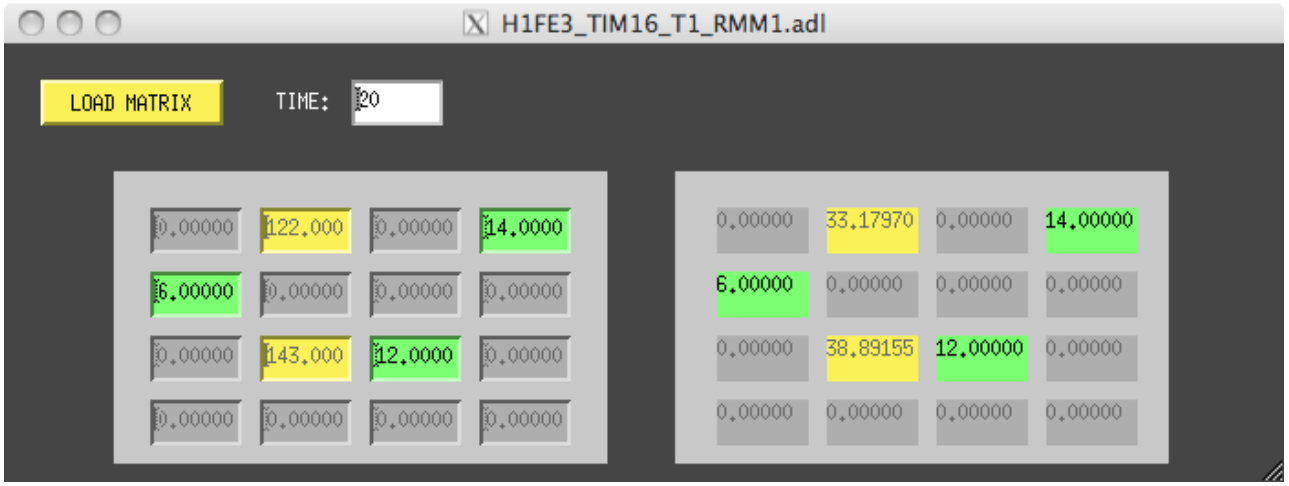
- Two matrix element display areas:
  - Left side allows entry of new values.
  - Right side shows matrix as presently loaded into the real-time code (readback channels).
- LOAD MATRIX: Settings entered by the user are not loaded to the real-time code until this load is executed. This allows the user to set in a series of new settings prior to them being passed to the real-time code.
- TIME: Time, in seconds, to ramp the values to be loaded into the real-time matrix from the settings once load is executed. This is a spline ramping function, identical to that used to ramp settings in IIR filter modules.

In this example, the first screen shot shows three (3) non-zero values in the settings area, which match the matrix read back channels from the RT code. All three settings/readings have a green background to indicate that they are identical.

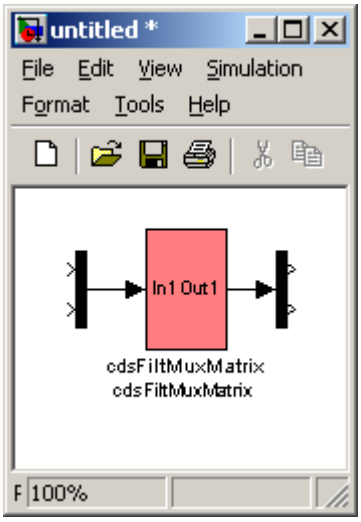


In the second view, two settings have been changed, and the read back channels do not match, as indicated by the red background. This is the case where new settings have been entered, but the LOAD has not yet been executed.

Once the LOAD is executed, the settings/read back channels will have yellow backgrounds while ramping is occurring, as shown in the following screen shot. Once ramping is completed, all non-zero settings/read back channels should go back to green.



7.9.5 cdsFiltMuxMatrix



7.9.5.1 Function

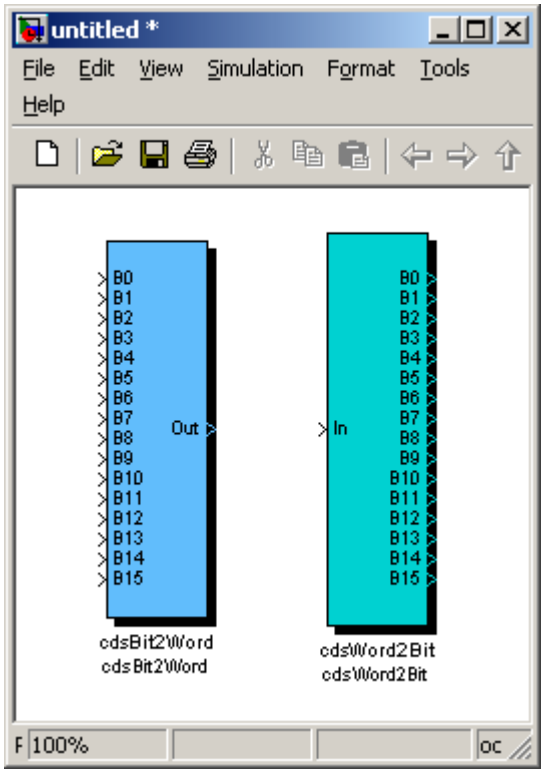
7.9.5.2 Usage

7.9.5.3 Operation

7.9.5.4 Associated EPICS Records

7.9.5.5 Auto-Generated MEDM Screen

7.9.6 cdsBit2Word/cdsWord2Bit



7.9.6.1 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.

7.9.6.2 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.

7.9.6.3 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.

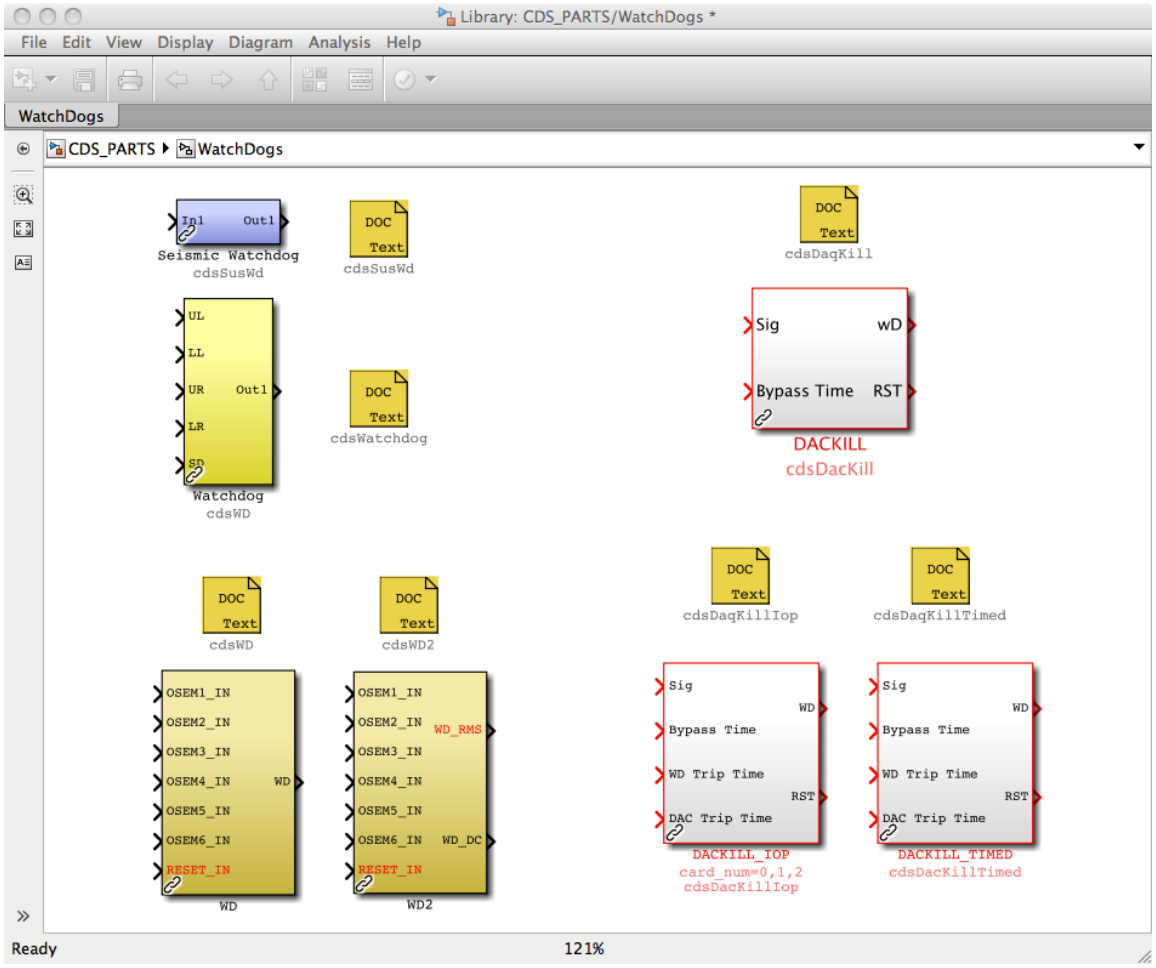
cdsWord2Bit will convert the 16-bit (integer) input, ‘In’, into 16 bits, e.g., the ‘In’ value 33,609 will result in the following bit pattern on the output: B15 = 1, B14 = 0, B13 = 0, B12 = 0, B11 = 0, B10 = 0, B9 = 1,

B8 = 1, B7 = 0, B6 = 1, B5 = 0, B4 = 0, B3 = 1, B2 = 0, B1 = 0, and B0 = 1.

7.9.6.4 Associated EPICS Records

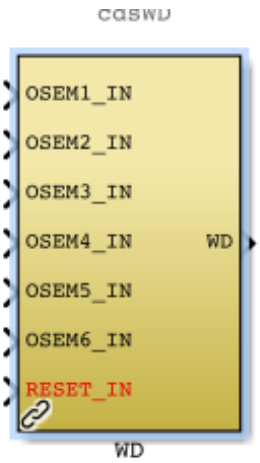
7.11 WatchDogs

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.



7.11.2 WD

This part was developed to provide watchdog protection for aLIGO large optics. A further description of this part and its usage can be found in LIGO-G1200172.



7.11.2.1 Function

This part is designed to monitor the six OSEMs of the top stage of a Quad suspension. It is built upon two subsystem SimuLink blocks, as shown in the following figures. This part is maintained with the CDS core code library to avoid alteration by application developers, as it is used for hardware safety purposes.

The first figure is a breakout of the top level. It shows the outputs of six individual OSEM monitors and how they are combined to produce a single watchdog trip output. The second figure shows the details of the individual OSEM monitor parts.

7.11.2.2 Usage

OSEM inputs 1 through 6 connect to the desired OSEM sensor signals. The RESET line provides a method to reset the filters associated with the DC and RMS filters used within the OSEM monitors. In practice, this is typically connected to the RESET output of a DACKILL part.

The output, WD, is a fault indicator, where 1=OK and 0=FAULT.

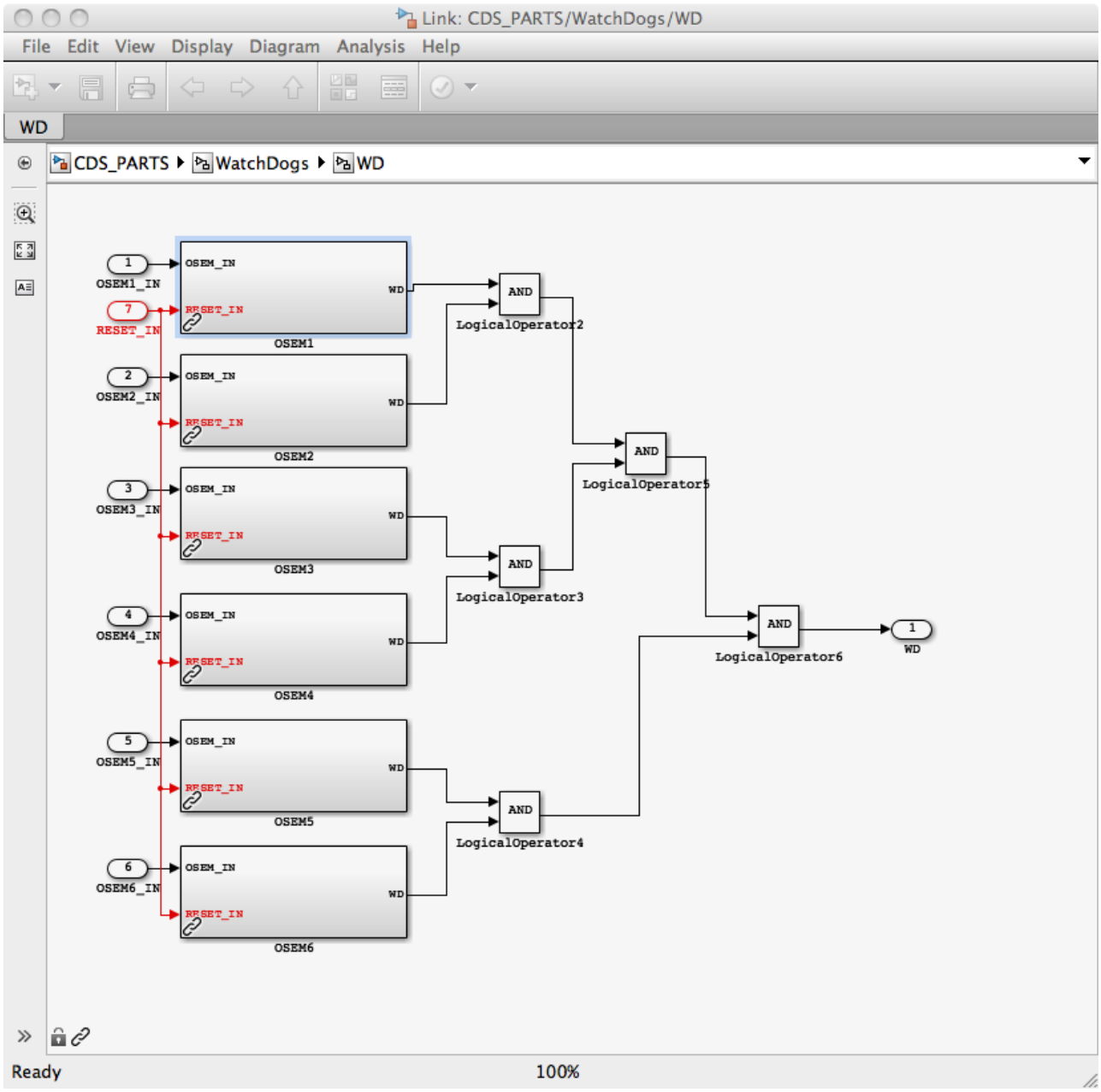


Figure 7: Six OSEM Monitoring Block

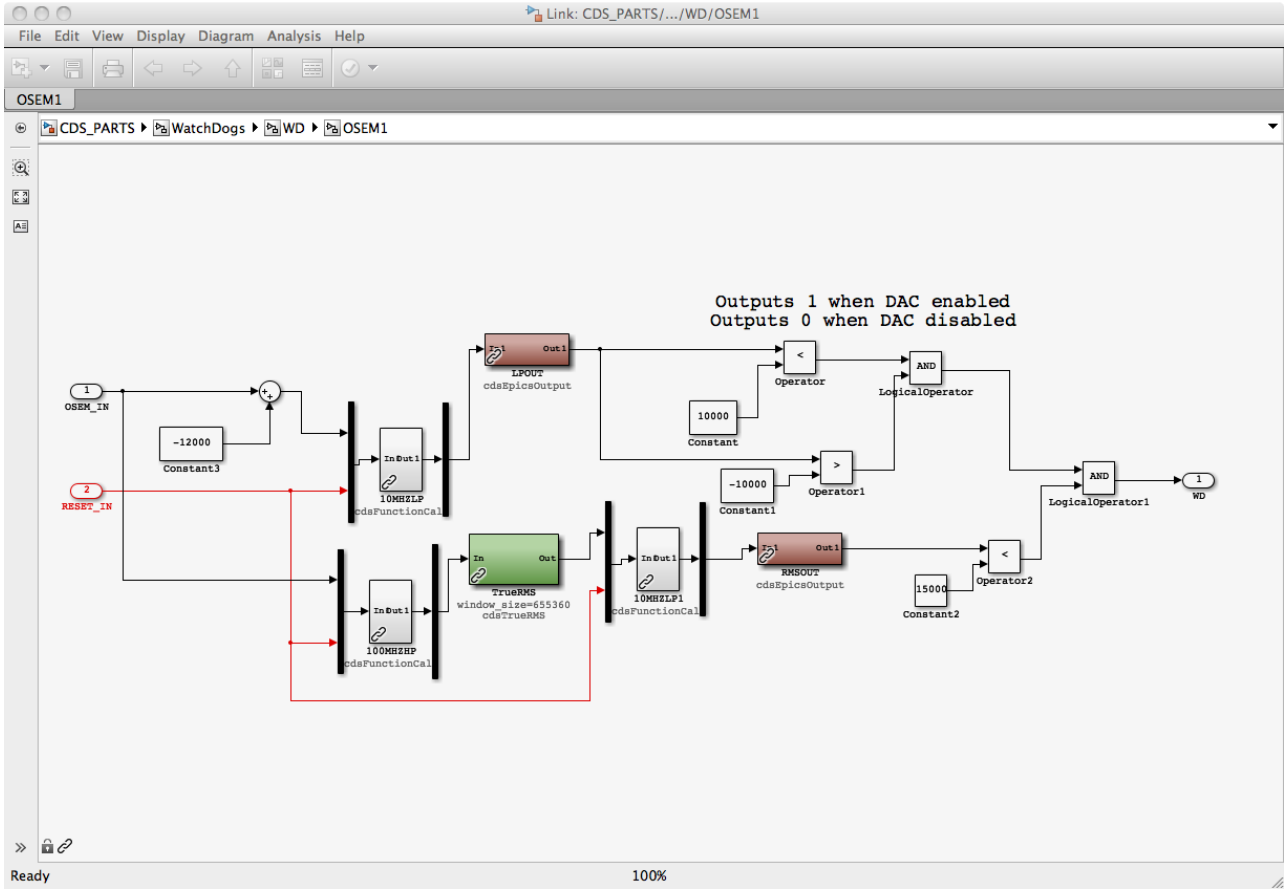
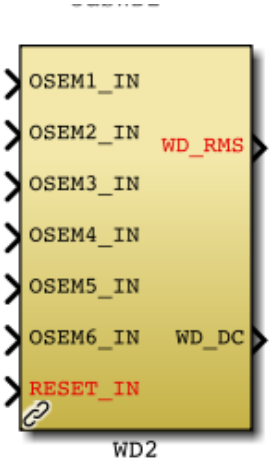


Figure 8: Single OSEM Monitoring Block

7.11.3 WD2



In preparation for a possible change in the suspension watchdog (SWD) chains, a variation of the original SWD part (WD2) has been added to CDS_PARTS. This new part has two outputs, instead of one:

- WD_RMS – The RMS Fault (0) or OK (1) output.
- WD_DC – The DC path Fault (0) or OK (1) output.

The purpose is to allow the application developer to determine which outputs to use to trigger fault indications and/or corrective action code, such as when connected to a DacKill part.

The internals of this part are shown in the following figure. Instead of ANDing together both DC and RMS paths, as done with the WD part, this part performs a separate AND for RMS and DC paths.

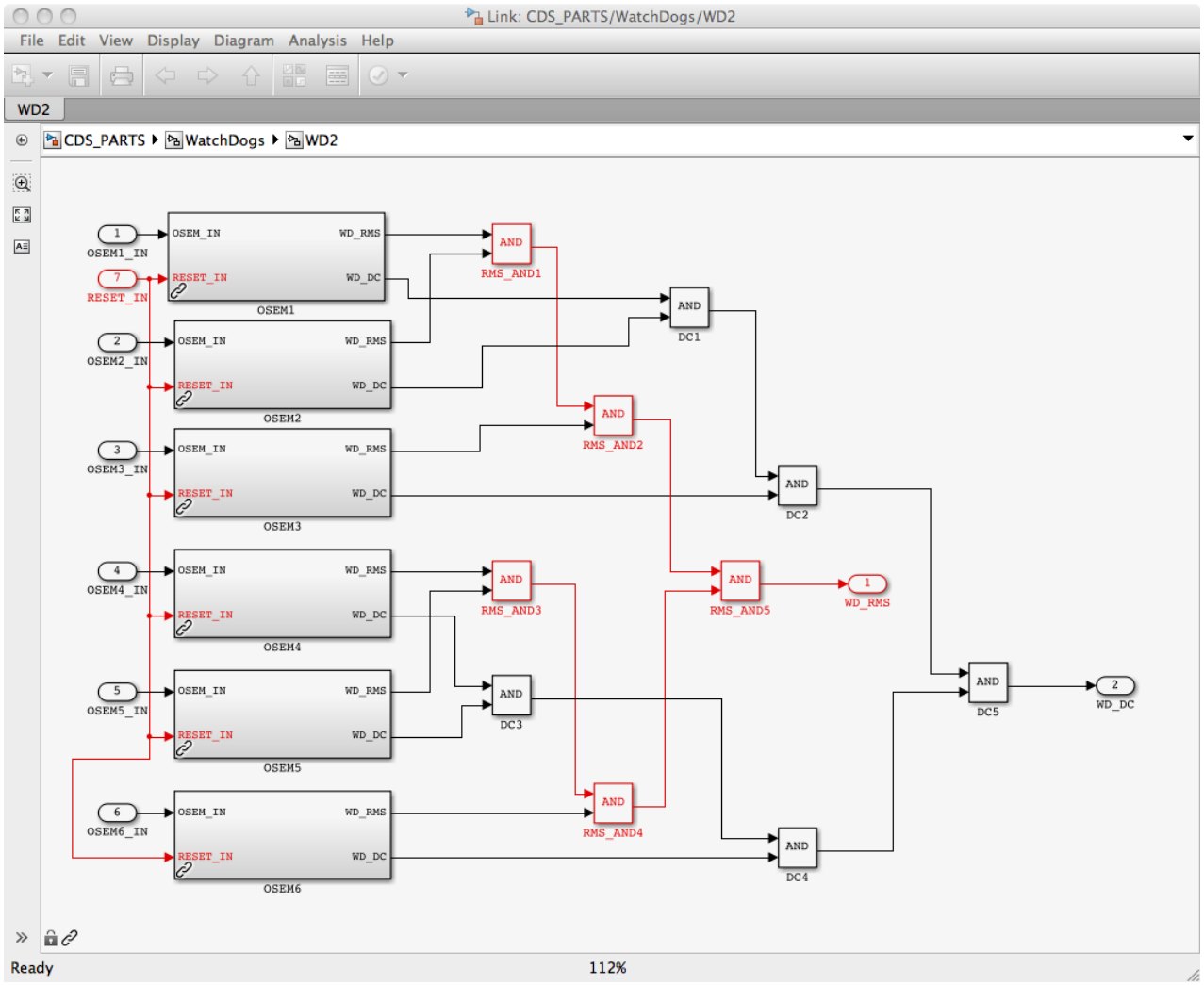


Figure 9: Generation of separate DC and RMS trip outputs

7.11.4 cdsDacKill

7.11.4.1 Function

The purpose of this part is force the code to output a zero (0) value to all DAC channels defined in the model, regardless of the actual application code requested value. This part typically receives a fault condition input from user specified fault monitoring logic/code within the RCG model.

NOTE: Only one (1) DacKill part may exist in a given RCG model.

7.11.4.2 Usage

This part has two inputs and two outputs, as described below. Input connections are required, but output connections are optional.

Inputs

1) Signal (0 = Fault, 1 = OK)
2) Bypass Time (Number of seconds WD can be bypassed)

Outputs

1) Watchdog Status (0 = Tripped, 1 = OK, 2 = Bypassed
2) Reset (Held HIGH (1) for one code cycle when WD reset. This output is intended for use within the user model to reset any fault detection code/logic.

7.11.4.3 Operation

This part has three defined states, as described in the following subsections.

7.11.4.3.1 MONITOR State

In this state, the code monitors the Sig input. As long as this input is one (1), all DAC outputs are sent as calculated by the user application. If the Sig input goes to zero (0), the code state will go to FAULT.

To achieve this state requires two things:

1) **Sig** input must be set to one (1)
2) After 1 above, a reset must be sent via the EPICS RESET channel (see next section).

On code startup, the default condition of the DacKill part is “FAULT”, and requires the above two conditions to clear the fault condition.

7.11.4.3.2 FAULT State

A fault state is entered when:

1) Application containing this part is first started, regardless of the **Sig** input value.
2) **Sig** input is zero and code is not presently in Bypass state.

3) Panic input is set to one via the EPICS PANIC input.

In this state, DAC outputs are set to zero. Which DAC channels are set to zero is dependent on the code model type:

1) IOP: All channels of all DAC modules connected to the computer will be set to zero.
2) User Application: Only those DAC channels defined in the user application will be set to zero. For example, if two user applications (app1 and app2) are sharing channels on the same DAC module, and the Sig input goes to zero only in app1, then:
   a. DAC channels defined by app1 will go to zero
   b. Those defined by app2 will continue to function normally

Note that once in this state, it will become "latched" ie even if the **Sig** input returns to one (OK), a RESET will be required to return to the MONITOR state. This state is also maintained as long as the PANIC input from EPICS is set to one.

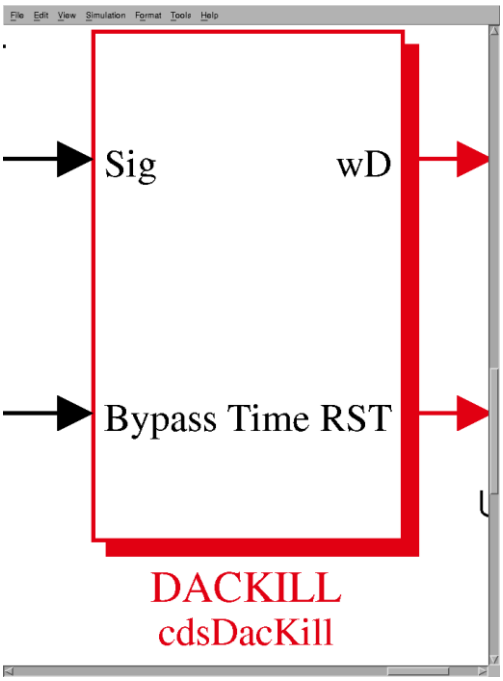
7.11.4.3.3 BYPASS State

Entering this state requires:

1) PANIC is not set to one, via the PANIC EPICS channel
2) BPSET EPICS channel momentarily set to one.

While in this state, the **Sig** input is ignored and all DAC channel outputs will continue to be passed normally from the user application code until either:

1) Bypass time expires. Note that once in the Bypass state, all further BPSET requests are ignored ie one cannot force reset of the Bypass timer and thereby extend the Bypass time. Once the timer has expired, the code will return to the MONITOR state (no RESET required).
2) EPICS PANIC is set to one. This will force the Bypass timer to be cleared and code to go to the FAULT state.



7.11.4.4 Associated EPICS Records

- Three EPICS Input Channels

1) **_RESET**: Momentary that:
   - a) Clears Trip State, if, and only if, Sig Input = OK
   - b) Turns OFF WD Bypass Mode
   - c) Sends 1 to RST output

2) **_BPSET**: (Momentary) Turns ON Bypass mode (all DAC outputs enabled) for number of seconds specified at Bypass Time input. During this time, the WD ignores Sig Input.
3) **_PANIC**: Binary input, trips and holds WD in a trip condition until PANIC turned OFF (0). Also clears BPSET, such that WD will not come back up in Bypass mode when PANIC turned OFF.

- Two EPICS Output Channels

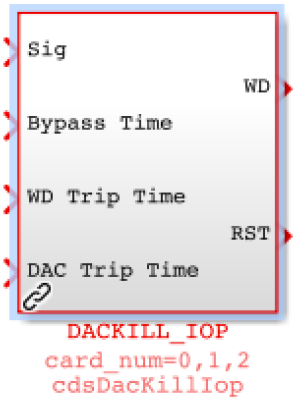
1) **_STATE**: The part output (wD) status:

a. 0 = Tripped (Fault)
b. 1 = OK
c. 2 = In BYPASS Mode

2) **_BPTIME**: Amount of time, in seconds, remaining on the bypass timer when in bypass mode.

7.11.6 cdsDacKillIop

7.11.6.1 Function



A new DACKILL part has been added to the library for RCG V2.8, DACKILL_IOP (DKIOP). This part is intended to more closely resemble the operation of the suspension system hardware watchdog system. It is presently only supported for use in IOP models.

Information on the motivations for the development of this part, which is an extension of the basic DACKILL (DK) part, can be found at https://awiki.ligo-wa.caltech.edu/aLIGO/SeiSusWatchDogCommissioning

7.11.6.2 Usage

This part has two inputs and two outputs, as described below. Input connections are required, but output connections are optional.

Inputs

1) Signal (0 = Fault, 1 = OK)
2) Bypass Time (Number of seconds WD can be bypassed)

Outputs

1) Watchdog Status (0 = Tripped, 1 = OK, 2 = Bypassed
2) Reset (Held HIGH (1) for one code cycle when WD reset. This output is intended for use within the user model to reset any fault detection code/logic.

7.11.6.3 Operation

7.11.6.3.1 Overview

The DKIOP part operates similar to the suspension system hardware watchdog. This part provides the ability to implement a two step process in attempting to clear a fault condition:

1) First trip: Set WD output to fault (0). This allows for the connection of code designed to take corrective action. In the particular case of the aLIGO SUS to SEI watchdog, this output is sent to the appropriate SEI controls IOP to shutdown HPI and ISI control outputs for that particular chamber, while allowing the SUS controls to remain operational.
2) Second trip: If corrective action taken by code initiated through connection to WD output has not cleared the problem, then shutdown the local DAC outputs for those DAC modules assigned to this DKIOP part.

7.11.6.3.2 Details

- A fault indication (0) must be present at Sig input consistently for the duration of time, in seconds, as set by the WD Trip Time input before any action is taken. If the fault indication is removed prior to this time, the timer is reset to start again on the next fault indication.
- Once the WD Trip timer expires:
    - WD output goes to zero (Fault). This output is latched in the fault condition ie a clearing of a fault indication at the Sig input at this point will not clear the fault nor reset the WD timer.
    - DAC outputs continue normal operation.
    - Second fault timer is started, based on time, in seconds, as defined at the DAC Trip Time Input.
        - Removal of a fault indication at Sig input will reset this timer to begin again on the next fault indication. Therefore, if the corrective action taken by code initiated by the WD fault output is successful, then the shutdown of DAC module outputs is prevented.
- Once the DAC Trip timer expires:
    - Outputs of all defined DAC modules are set, and latched, to zero.
    - Removal of fault indication at Sig input will not clear this, nor the WD fault output.
        - Requires DKIOP Reset command and clearing of fault indication at Sig input to clear the trip condition.

7.11.6.3.3 Example Use in SUS to SEI Watchdog

As an example of how the DKIOP might be used in the h1iopsush34 model, connecting to the h1iopseih23 model, is shown in the following figures.

In Figure 1, a portion of the h1iopsush34 model is shown. The suspensions in HAM3, MC2 and PR2, have their watchdogs connected to DKIOP HAM3. The WD output of this DKIOP would be connected to an IPC part (not shown) to send the signal to the seismic system IOP. The SR2 suspension, located in HAM4, has its watchdog connected to DKIOP HAM4.

Figure 2 shows the DKIOP components of h1seih23. The input to its DKIOP HAM3 would be received via the IPC connection made in the SUS IOP.

Given the parameters shown in this example, the shut down sequence would be:

- In h1iopsush34 controller:
    - On fault indication from either PR2_WD or MC2_WD, the DKIOP HAM3 would start its WD timer (set to 600 sec, or 10 minutes, in example).
    - If fault exists beyond the 10 minutes, the WD output will go to fault condition (0 output), which is sent to via IPC to DKIOP HAM3 in seismic system IOP.
- In h1iopseih23 controller:
    - After 1 second, the HAM3 WD output of the seismic system IOP will go to fault (0).
    - After 2 seconds, the HAM3 DKIOP of seismic system will disable all DAC channels outputs to DAC module 1. This, in turn, will shutdown the HEPI and ISI controls to HAM3.
- In h1iopsush34 controller:
    - If the shutdown of HEPI and ISI in HAM3 has not resulted in the clearing of fault signals from either PR2 or MC2 within 900 seconds (15 minutes) after the DKIOP

HAM3 WD output was set to fault, then zero out the drives to the this model's DAC card 0.

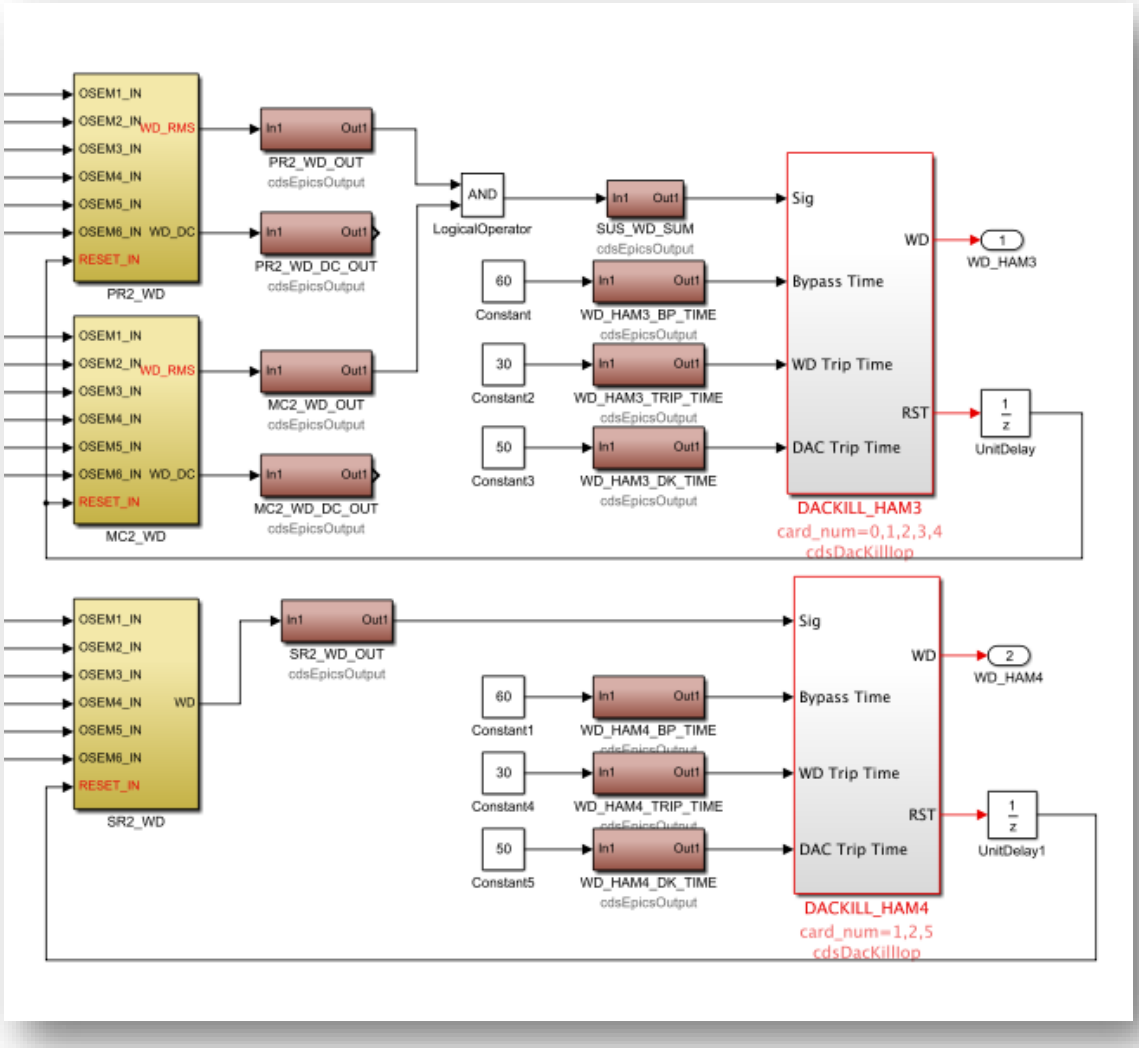


Figure 10: Example h1iopsush34 model

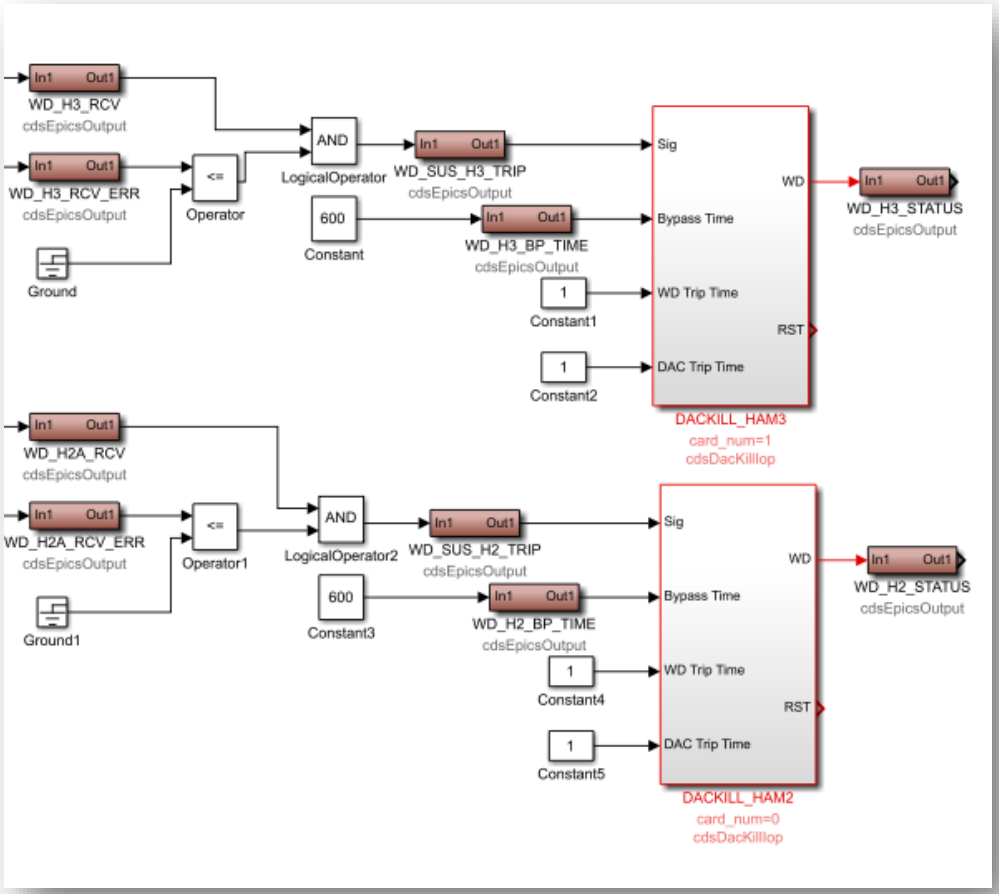


Figure 11: Example h1iopseih23 model

7.11.6.4 Associated EPICS Records

- Three EPICS Input Channels

1) **_RESET**: Momentary that:
   - a) Clears Trip State, if, and only if, Sig Input = OK
   - b) Turns OFF WD Bypass Mode
   - c) Sends 1 to RST output

2) **_BPSET**: (Momentary) Turns ON Bypass mode (all DAC outputs enabled) for number of seconds specified at Bypass Time input. During this time, the WD ignores Sig Input.
3) **_PANIC**: Binary input, trips and holds WD in a trip condition until PANIC turned OFF (0). Also clears BPSET, such that WD will not come back up in Bypass mode when PANIC turned OFF.

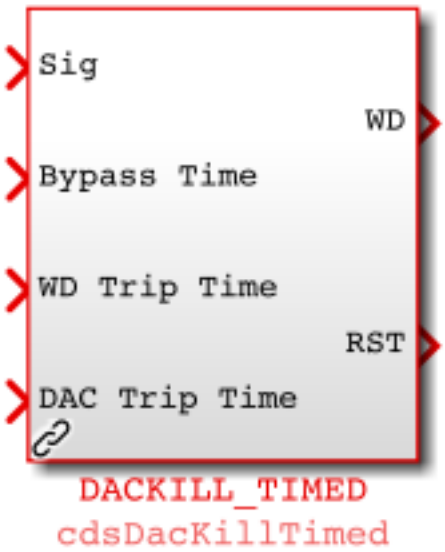
- Two EPICS Output Channels

1) **_STATE**: The part output (wD) status:

a. 0 = Tripped (Fault)
b. 1 = OK
c. 2 = In BYPASS Mode

2) **_BPTIME**: Amount of time, in seconds, remaining on the bypass timer when in bypass mode.

7.11.8 DacKillTimed



The functionality of this part is similar to the DacKillIOP part. The primary differences are:

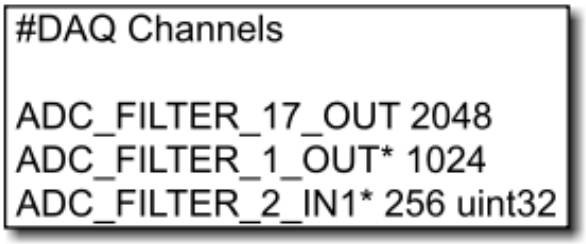
1) It may be used in either an IOP or user application model.
2) Like the original DacKill part, it will cause the shutdown of all DAC channels defined in the model ie does not have DAC module assignment capability that the DacKillIop part does.

Item 2 actually makes it more suitable for use within a user model, as it will direct the DAC outputs to zero only for those channels used by the model instead of entire DAC modules.

7.13 DAQ Parts

7.13.1.1 Function

The function of this part is to define model channels that are to be sent to the DAQ system for data storage.



7.13.1.2 Usage

This part may be placed at any level within an RCG model.

7.13.1.2.1 RCG Releases prior to RCG V2.6

Entries must be made as:

ChannelName AcquisitionRate

All data channels are recorded as 32bit floating point type.

7.13.1.2.2 RCG V2.6

In V2.6, support was added for acquiring unsigned int (UINT32) type data. Channels to be recorded as this type must have an additional entry added, as shown above (uint32).

7.13.1.2.3 RCG V2.7 and later

In V2.7, support was added to record two types of data frames simultaneously:

- Commissioning frames: Contain all of the data indicated in the DAQ Channel part.
- Science Frames: Contain only those channels indicated by an asterisk (*) at the end of the name, such as ADC_FILTER_1_OUT* in the example above.

Note that all EPICS channels are still recorded in both frame types.

The reason for two frames was to allow commissioners to record more data channels in the short term on CDS local disk, necessary for commissioning activities, but not important for long term data archive and scientific analysis. In this manner, it is hoped to keep the long term storage costs down.

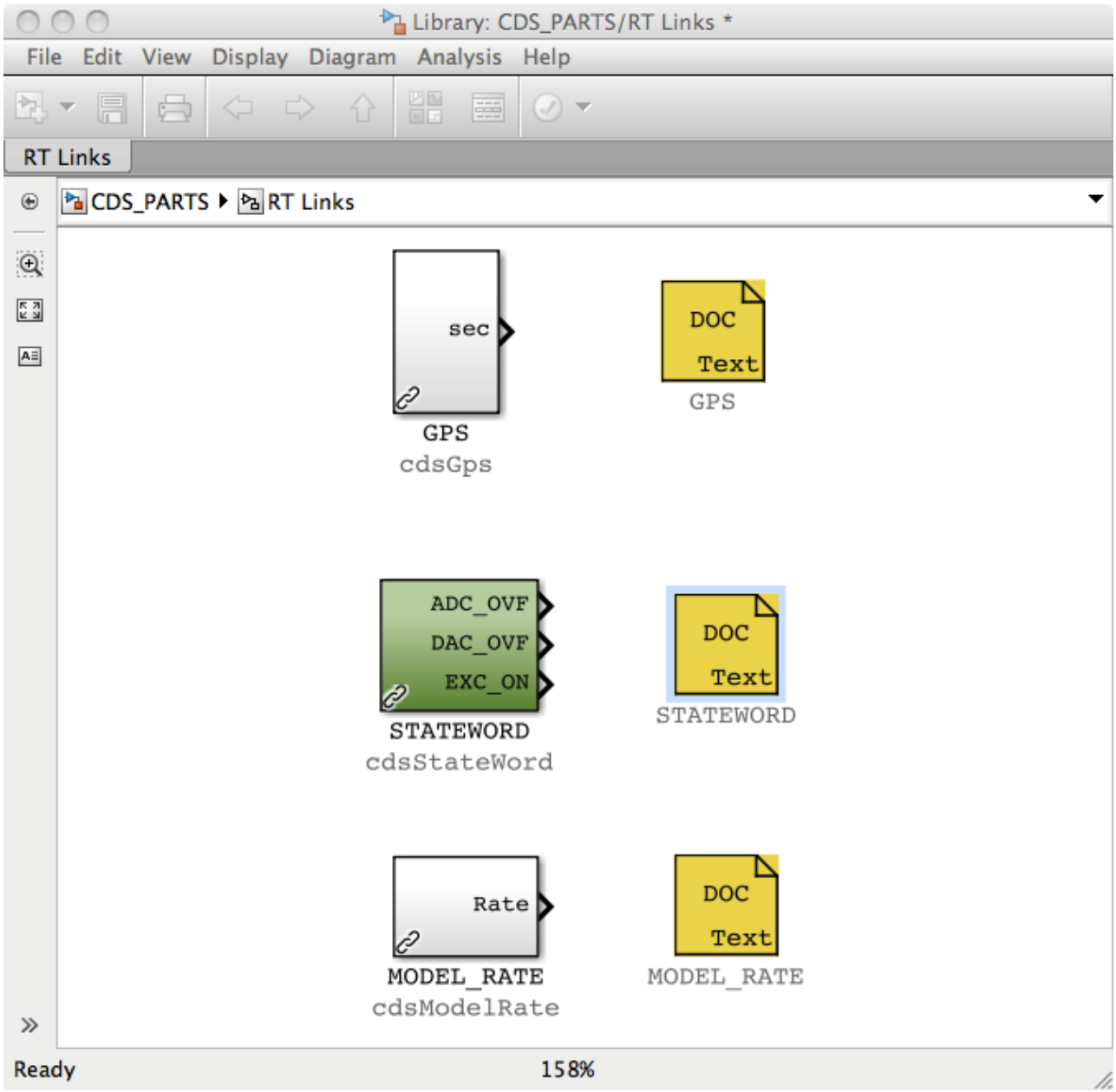
7.13.1.3 Operation

During the code installation process, these channels will be set within the MODELNAME.ini file to acquire data at the desired rate. This file is used by both the RT runtime code and the DAQ system to determine which channels are to be recorded and at what rate.

The UINT data type was added primarily to support recorded of system state words. In V2.6, down sampling of UINT32 type data is done by simply recording every nth sample. Beginning with V2.7, down sampling is done by performing an AND function. For example, if recording rate is set to 256 and code runs at 2048, an AND is performed on 8 samples for each sample recorded. In this manner, it is possible to capture those bits which may have changed over the number of range of down sampled values.

7.14 RT Links

In RCG V2.8, a new subsystem block has been added to the CDS_PARTS.mdl. This subsystem block contains parts which provide access to values already calculated by the real-time code, but formally unavailable to the real-time application.

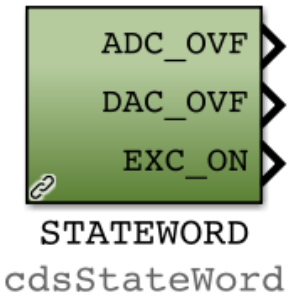


7.14.1 GPS

A part has been added to provide the current GPS time, in seconds, at the output.

7.14.2 ODC State Word

To further support On-line Detector Characterization (ODC), a part is provided that outputs a set of status information. This part has three, single bit outputs, updated on every real-time code cycle.



- ADC_OVF = Overflow condition exists on one, or more, ADC channels being used by the application.
- DAC_OVF = Overflow condition exists on one, or more, DAC channels in use by this application.
- EXC_ON = One, or more, GDS Excitation signal is actively injecting a signal to this application. Note that this bit is not set if the excitation signal channel is simply being read, as by Dataviewer.

7.14.3 Model_Rate

This part outputs the defined rate at which the code is running eg 2048, 16384, etc.