LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY

LIGO Laboratory / LIGO Scientific Collaboration

LIGO-T0900607-v5 *LIGO* May 4, 2012

aLIGO CDS
Real-time Sequencer Software

R. Bork/A. Ivanov

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

All LIGO Control and Data System (CDS) real-time control and monitoring tasks must run synchronously at 2^n rates, from 2048Hz to 65536Hz. The purpose of this document is to describe the real-time software which performs this synchronization task.

2 References

1) AdvLigo Timing System Document Map
2) aLIGO CDS Design Overview LIGO-T0900612
3) CDS Realtime Code Generator (RCG) Application Developer's Guide (T080315)
4) CDS Inter-Process Communications (IPC) Software Design LIGO-T1000587
5) CDS Real-time Data Acquisition Software LIGO-T0900638.
6) RCG Runtime Diagnostics V2.4 LIGO-T1100625.
7) aLIGO CDS Timing Control LIGO-T1000635.

3 Overview

The aLIGO CDS design includes a number of computers, in a distributed computing architecture, to perform real-time (deterministic) control and monitoring and data acquisition tasks. All of these real-time tasks are designed to run synchronously at 2^n rates, from 2048Hz to 65536Hz.

The primary hardware components of this system, as shown in the following diagram, are:

1) Muli-core computers running real-time applications, with various network connections and PCI Express (PCIe) fiber link to a remote I/O chassis.
2) Input/Output (I/O) chassis, which contains various Analog-to-Digital and Digital-to-Analog Convertors (ADC/DAC) and Binary I/O modules.
3) aLIGO Timing Distribution System (TDS).

More information on the hardware components can be found in References 1 and 2.

Each real-time computer, commonly referred to as a Front End Computer (FEC), is intended to run multiple real-time applications. Each real-time application is defined and compiled using the CDS Real-time Code Generator (RCG), as described in Reference 3. This executable code build includes two primary components:

1) CDS core software: A set of standard functions and libraries which act as a wrapper for all user specified applications, with the key components of:
    a. Real-time Sequencer (RTS), the main focus of this document, that has primary responsibility for timing and synchronization. Source code is located in the CDS SVN as *trunk/src/fe/controller*.c.
    b. I/O Drivers: Software used to communicate with hardware devices. A basic overview of the key elements of this code is discussed in this document, as it relates to the RTS. Source code is located in *./trunk/src/fe/map.c*.

c. Inter-Process Communications: All processes within a single computer and/or connected on the CDS real-time networks are able to synchronously communicate with each other. Details of this design are provided in Reference 4.
d. Real-time Data Acquisition: This is described in detail in Reference 5.

2) Specific user application: Source code developed by the RCG at build time.

All elements are compiled into a single code thread, as a Linux kernel object. Depending on compile options, this code is set to run as a Master, commonly and hereafter referred to as an I/O Processor (IOP), or a Slave (user application). Each FEC is set up to run a single IOP and multiple slave applications, as shown in Figure 1. These are described in detail in the remainder of this document.

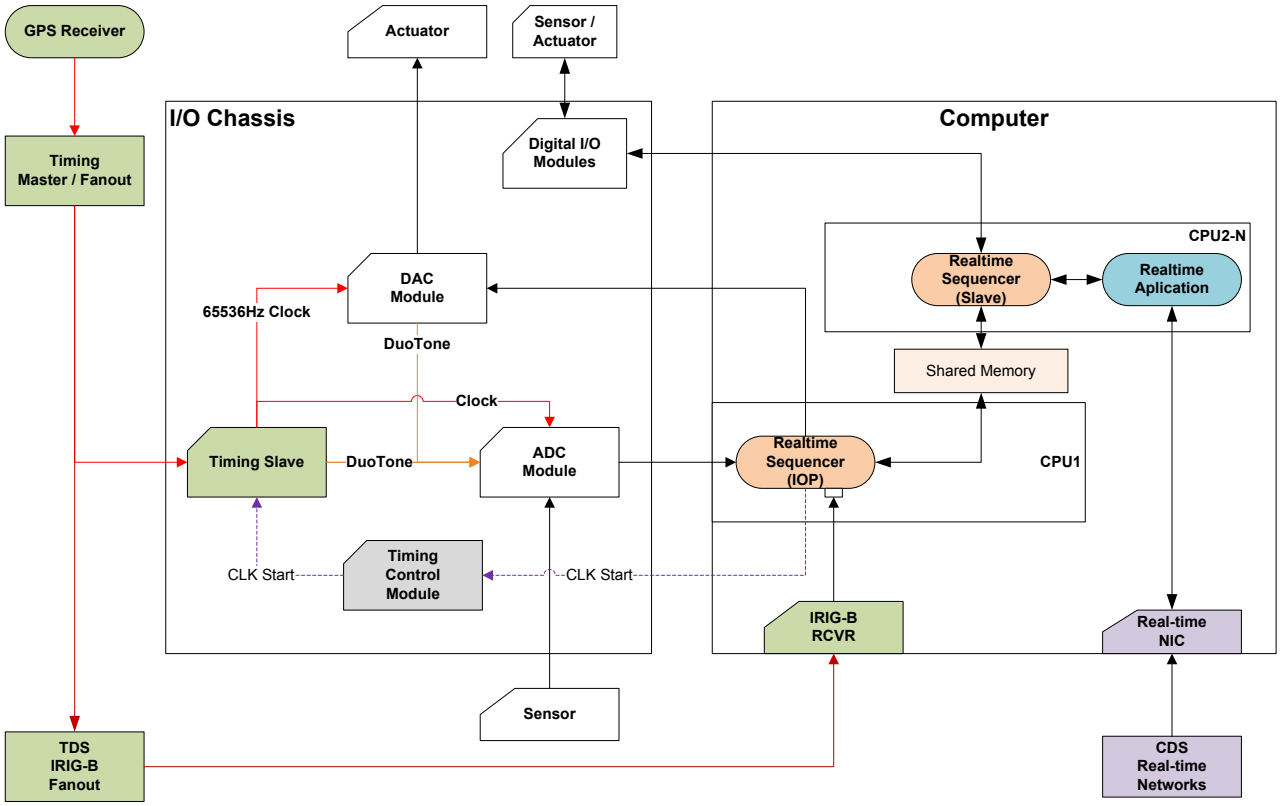


Figure 1: CDS Real-time System Overview

4 Operating System

The present operating system loaded on to FEC is gentoo Linux, kernel release 2.16.34. However, even with real-time extensions, this General Public License (GPL) Linux, and its task scheduler, cannot meet the "hard real-time" requirements of the CDS real-time systems. CDS requires deterministic behavior on the order of µsec. Standard Linux schedulers operate on the order of

hundreds of μseconds, and even the PREEMPT real-time scheduler, available as part of GPL Linux, is only deterministic to order tens of μseconds.

To achieve the "hard real-time" level of deterministic behavior (few μsec) requirements of the FEC software, the CDS real-time implementation does a couple of things:

- Linux OS and scheduler are limited to running non-realtime critical support tasks, such as the EPICS interface to real-time tasks (see Section 4 of Reference 2), and as a tool to load and unload real-time tasks.
- All real-time tasks are assigned to and locked into individual CPU cores.
- Execution of real-time tasks are "scheduled", ie triggered, by arrival of an ADC sample. Since the ADC modules are all clocked synchronously by the TDS, these tasks are now also synchronous with the TDS.

In order to lock CDS code into a CPU core and divorce itself from the Linux scheduler, a patch was developed by CDS to compile in with the standard GPL Linux kernel. This is a simple patch, which functions as follows:

- When the CDS real-time application is loaded, as a kernel object, a call is made to the standard Linux CPU shutdown function. This function takes the CPU core offline and removes it as a resource from the Linux OS scheduler ie Linux no longer knows the CPU core is there and will not try to schedule tasks or send interrupts to it.
- Without the CDS patch, after CPU shutdown, a "do nothing" idle task is invoked by the Linux shutdown procedure for this CPU. The CDS patch essentially replaces this idle call with a call to load and run the CDS real-time application.

The end result is that the CDS real-time code is now running in isolation, free of any Linux OS operations or scheduling, and strictly slaved to the ADC modules to trigger its operation.

The Linux patch itself is maintained in the CDS core software repository. This patch was line-by-line reviewed by two experts (Linux device drivers and real-time extensions) from the Linux consortium at a meeting in Hannover, Germany in July, 2010 to verify that it was properly implemented.

5 Operational Modes

The RTS is designed to operate in three modes, provided as compile options in the RCG:

1) Stand-alone: The RTS handles all of its own I/O by directly reading/writing PCIe I/O devices. This was the standard design prior to release 1.9 of the RCG.
2) Master, commonly referred to as the Input/Output Processor (IOP): In this mode, the RTS does not run an associated real-time control application. Its purpose is to handle all ADC/DAC I/O and timing for all other applications running on the same computer.
3) Slave: The RTS runs "slaved" to the IOP for timing and ADC data.

The "stand-alone" mode is being phased out in favor of the single IOP, multiple Slaves per computer model. This is driven by:

1) Timing system and timing interface design in I/O chassis. To provide for startup of the system synchronous with the GPS 1PPS mark, the timing slave in the I/O chassis provides a gating capability to start its clock outputs on the 1PPS mark. Once the clock is started, all ADC modules are triggered to sample, and continue sampling on every clock. Therefore, there must be a task to both:
    a. Configure and initialize all I/O modules prior to start of the clocks.
    b. Gate the timing slave to start the clocks.
    c. Be available to receive data from all ADC modules and setup their next data transmission. Without this, the ADC FIFOs would quickly overflow and operation would stop.
2) Requirement to be able to have multiple applications share ADC channels from the same ADC module and, conversely, write to individual channels of shared DAC modules. This is handled by the IOP by placing ADC data into shared memory, and reading DAC data from shared memory, which is read by/ written to by slave applications. This is not provided for in the Stand-alone compiled version.

6 IOP Code Model

The IOP is defined in a standard Matlab model and compiled by the RCG in the same fashion as any other user real-time application. It is intended that there be a single, generic IOP model, from which all other IOPs are developed. An example is shown in the following two figures, with the first being the top level model view and the second showing the internals of the "MADC" subsystem components.

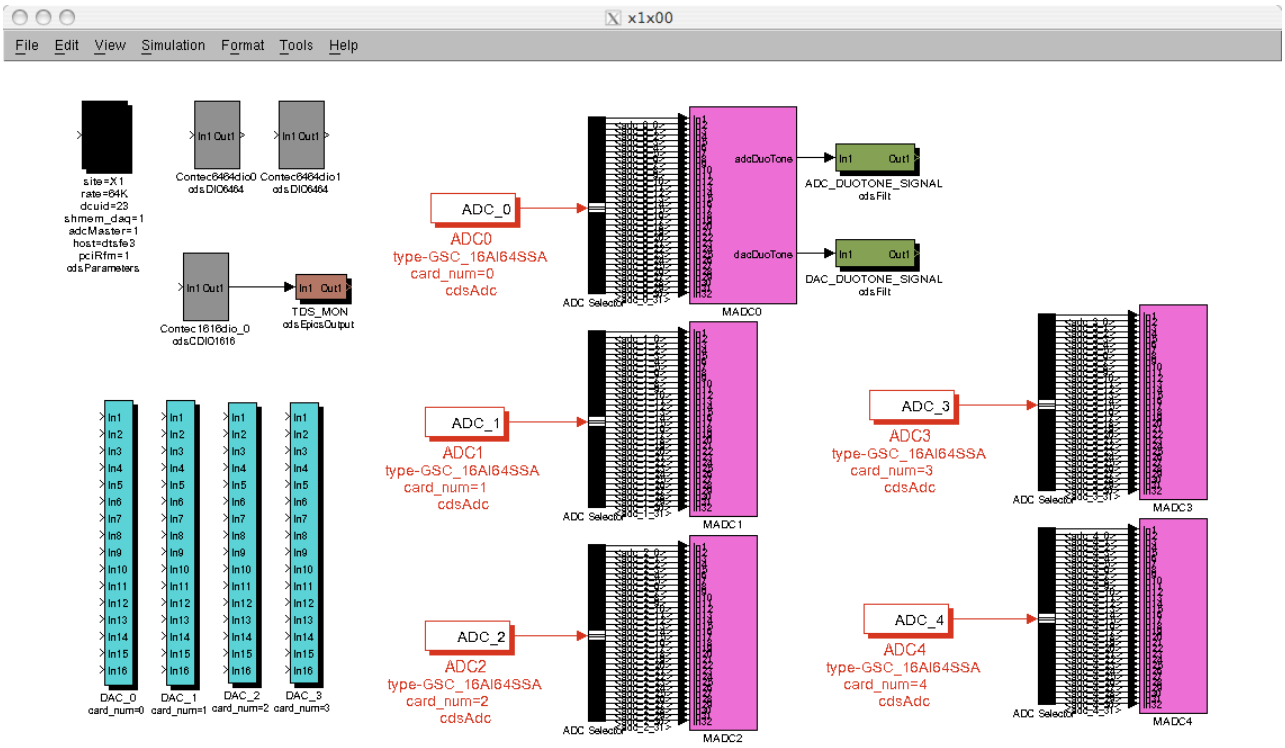
This model almost exclusively contains I/O components. For every I/O module to be installed in the connected I/O chassis, an appropriate I/O part is placed in the IOP model. In this example, there are five ADC modules and 4 DAC modules, along with two binary I/O cards.

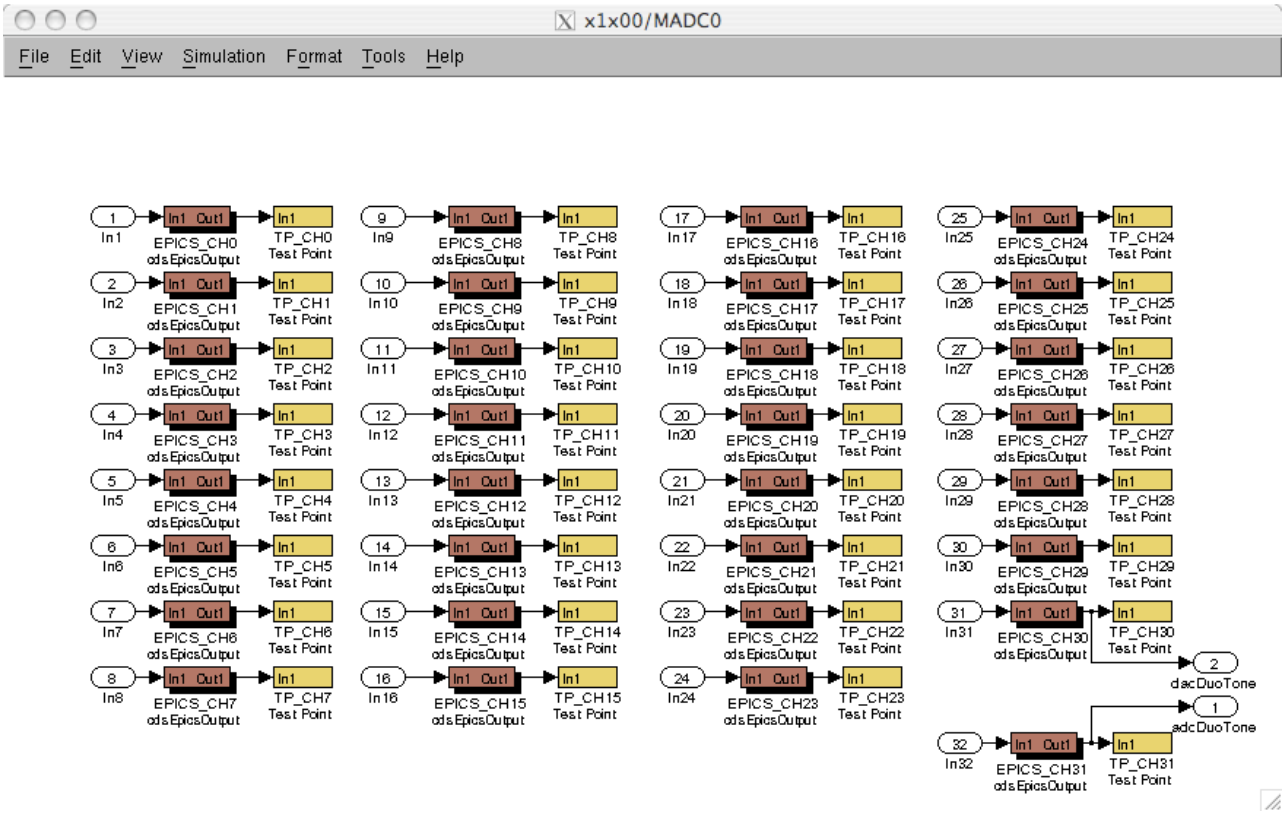
For each ADC module, there are EPICS channels and GDS Test Points defined, which are provided as diagnostics. There are also two filter modules attached to the first ADC, last two channels. These are provided for acquiring the ADC and DAC duotone channel data, which provide timing diagnostics.

There are also two other significant components in this model:

1) Parameter Block (Upper left of first figure). This block, found in all RCG models, is used to define RCG compile options. For this model to be compiled as an IOP, the parameter block must contain the following parameter lines:
    a. adcMaster=1: Indicates model to be compiled as IOP. Conversely, all applications which are to run as slaves to an IOP must contain "adcSlave=1". If neither is defined, then the application is compiled as a "stand-alone".
    b. rate=64K: IOP is designed to run at 65536 samples/sec.
    c. pciRfm=1 (Optional): If the IOP is to run on a computer connected to the Dolphinics real-time network, this flag must be set to load the appropriate drivers. (NOTE: This should only be set in IOP models, never in Slave application models, even if those models make use of this network. If network is available, the IOP will pass this information along to the slave processes.)

2) Contec1616 binary I/O part: This card is used by the IOP to control the TDS timing slave in the I/O chassis.





7 IOP Runtime Execution

Execution of the IOP is invoked by installing the IOP kernel module, produced by the RCG during the compile phase. Once loaded, this module:

1) Performs various initialization tasks, as outlined in Section 8.1.
2) Executes in a continuous loop, each loop 'cycle' triggered by receipt of ADC data. Execution is further described in 8.2.

7.1 Initialization

A number of items must be taken care of during the code initialization phase:

- Establish pointers to the shared memory blocks used for communications with EPICS, DAQ and AWGTPMAN.
- Establish pointers to shared memory blocks used to communicate data with real-time slave processes.
- Find, map and initialize all PCIe hardware to be used by the real-time code.
- Pass I/O pointers, via shared memory, for use by slave processes. Note that while the IOP will provide all direct communication with ADC/DAC modules itself, the slave processes are responsible for direct communication with any binary I/O or network interfaces.

7.1.1 Shared Memory

Computer shared memory is used to communicate data between real-time applications and their non-real-time support tasks (EPICS/AWGTPMAN/DAQ) and to communicate data between each other. The blocks created for each real-time process, including the IOP, are:

- EPICS shared memory area, for communication with EPICS.
- DAQ shared memory area, for communications with AWGTPMAN and DAQ network driver software.

In addition, per computer, the IOP initializes an Inter-Process Communication (IPC) shared memory block. This area is used by all real-time processes to:

- Communicate I/O pointers and other I/O device information from the IOP to slave processes.
- Pass ADC and DAC data between the IOP and slave processes.
- Communicate real-time data between real-time tasks (See Reference 4).

7.1.2 I/O Mapping

All I/O mapping and initialization is performed by the mapPciModules() function. Source code for this function and most other I/O functions is located in the CDS SVN source code file ***./trunk/src/fe/map.c***.

For each I/O device defined in the IOP model, this function attempts to find that module on the PCIe bus. If a module is found, it calls the appropriate initialization routine. Every initialization routine includes:

1) Enabling of the pci device (pci_enable_device).
2) Remapping of device I/O registers to CPU memory for access by applications.
3) Placement of memory locations/pointers in a standard CDS structure for later use by the IOP and slave applications.

7.1.3 ADC Initialization

The ADC modules employed in CDS systems have 32 individual ADC channels, with 16 bit resolution. All ADC modules are clocked at 65536Hz, a rate chosen for the optimal noise performance of the ADC modules used by CDS.

To enhance I/O performance, these modules have a capability known as “Demand DMA Mode”. In this mode, whenever an ADC FIFO contains >= the user defined number of samples, and the “DMA Start Bit” has been set, the ADC will automatically transfer the defined number of samples to the user specified computer local memory location. This is the mode that the CDS code uses, with initialization shown in the following flow diagram.

A couple of items of note:

1) ADC data is transferred as a 32 bit integer per channel, with the lower 16 bits containing the data.
2) The first channel is tagged, by the ADC, by bit 17 being set. For all other channels, no upper bits should ever be set.
3) Once in a run mode, the code will only read data from local memory (ADC does the data transfer automatically in Demand DMA Mode).
4) Given 3 above, the initialization routine writes a zero into the local memory channel 0 location and 0xf000000 into the channel 31 location. If operating properly, the ADC will never write these values to these locations ie channel zero should always have an upper bit set and channel 31 should never have upper bits set (above the 16 bit data).
5) Once the ‘DMA Start Bit’ is set, the ADC will automatically transfer 32 channels of data, from its FIFO, for each 65536Hz clock received from the timing slave.

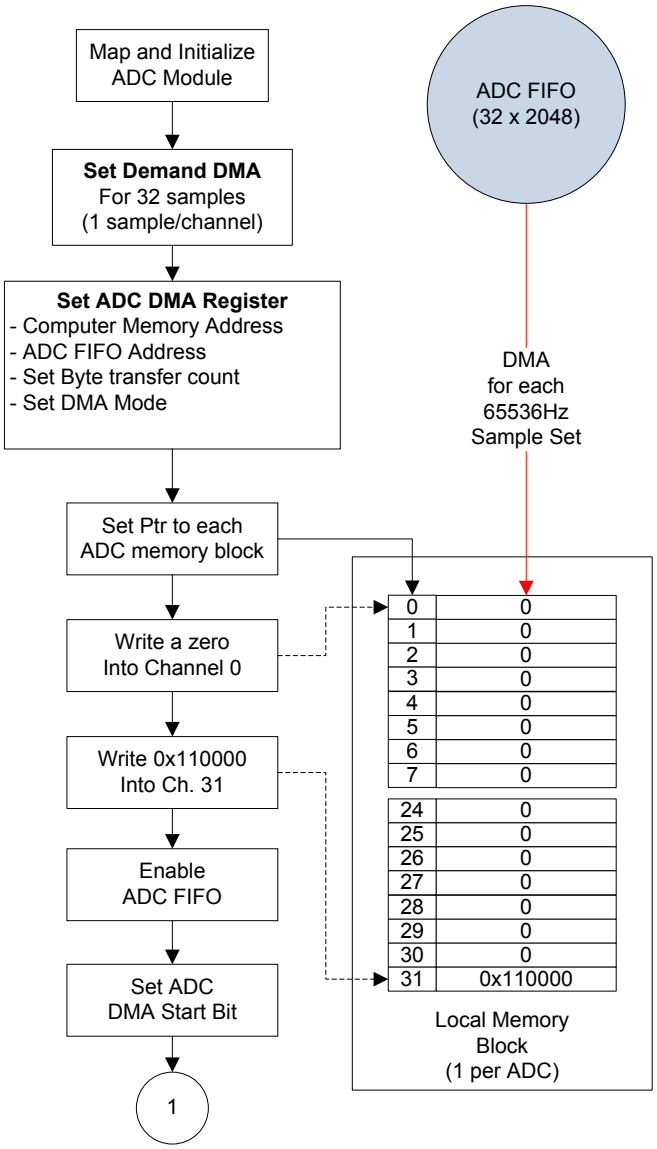


Figure 2: ADC Initialization[1]

7.1.4 DAC Initialization

Once the ADC modules have been initialized, then the DAC modules are initialized, as shown in the following flow diagram. As part of the initialization, the DAC module DMA register is preset with the number of bytes to read on each DMA request and read/write memory locations. In this fashion, whenever the IOP is ready to send data to the DAC, it is simply a setting of the "GO" bit in the DMA register. This is done in the interest of saving time during execution ie not necessary to write all of the DMA information on every code cycle. This method also makes use of the DMA

[1] Note: Diagram shows setting of 0x110000 into ADC channel 31. This was changed to 0xf000000 in order to also support 18bit ADC modules.

engine on the DAC board, which actually pulls the data from CPU memory with no code CPU clock cycles required.

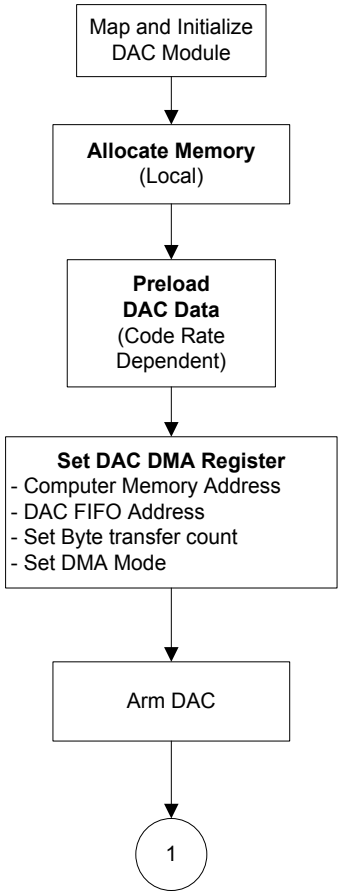


Figure 3: DAC Initialization

7.1.5 Remaining I/O Initialization

For remaining I/O modules, the initialization is simply enabling and mapping the device. Since the slave application processes will directly access these devices, such as binary I/O and real-time network cards, the IOP must pass this information along to the slave processes. Also passed are the ADC/DAC card information, including pointers to the IPC shared memory to be used for transfer of ADC/DAC data. This is done via a reserved area within the IPC shared memory block.

7.2 Startup Synchronization

7.2.1 Sequence

Once initialization is complete, the IOP must start its infinite loop synchronous with a GPS 1PPS time mark. This is done by:

1) Disabling the timing clocks from the timing slave.
2) Enabling the Demand DMA mode on the ADC modules, along with clearing their FIFOs.
3) Preload the DAC module FIFOs. The DAC FIFOs are used as a buffer for two reasons:
    a. Allows room for IOP "timing jitter". If IOP needs to deal with many DAC modules, it is possible that writing of some modules may come before the DAC clock and some after, resulting in data output timing differences between the modules. Preloading prevents this by ensuring at least one sample is ready on every DAC in the IOP output time frame.
    b. Use as a diagnostic. Once per second, the IOP checks the FIFO data size to verify that there is one, and only one, sample remaining.
4) Enabling the timing slave Gate signal.

Once the latter is done, the timing slave will start generating clock outputs coincident with the next 1 second time mark and run continuously thereafter.

7.2.2 Timing Control Registers

The timing slave is sent control signals via a Contec binary I/O module and the I/O chassis timing interface backplane. This backplane routes timing clocks between the timing slave and the ADC/DAC modules via ADC/DAC interface cards inserted into this backplane. One clock is used to run all ADC modules in the IOC and one to run all DAC modules. This backplane also provides an interface to the duotone signal from the timing slave to the last channel of the first ADC module in the IOC for timing diagnostics.

A couple of notes:

- Read/writes are performed with an unsigned 32 bit integer. The lower 16 bits are read and upper 16 bits are write. Register definitions for the timing control binary I/O module are listed in Reference 7.
- The first ADC and DAC module interface cards have relays which allow:
    - Duotone from timing slave to be fed into first ADC card.
    - Output of first DAC, last channel to be connected back to first ADC card with intent of feeding back the duotone read in the ADC and measuring loop time delay.
    - Timing slave provides two gated clock outputs: one which goes positive in coincidence with the 1PPS mark and one which goes negative. The IOC timing interface backplane allows selection of the positive or negative edge clock to the ADC and DAC modules.
    - The DAC clock polarity is set opposite of the ADC clock in the default setup. This is further discussed later in this document.

7.3 IOP Processing Loop

Once the timing clocks have been enabled, the IOP process begins an infinite loop, as shown in the following figure. A few items of note:

1) The IOP runs at the aLIGO timing system clock rate of 65536Hz, and therefore must complete its processing loop in ~15μsec.
2) The most time critical components are the ADC read and DAC write. The completion of ADC reads triggers the slave applications to run, so must be kept to as minimal amount of time as possible.
3) The IOP polls for data available from the ADC modules to trigger the loop, as indicated by a change in the ADC channel 31 memory location. (There is another kernel method to have the CPU hardware notify the task when data in a specified memory location has changed. This code is, at least temporarily, backed out as this feature is not available on all CDS computers presently in use ie non-Intel based computers).
4) IOP rearms the ADC DMA Start Bit. This enables the ADC module to send data again once it has a sample ready.
5) The IOP writes a GPS timestamp and 65K cycle count along with the ADC data to shared memory. The time and cycle are then inherited by the slave (user) applications, after first verifying that this was the expected time and cycle count.
6) After ADC values read, IOP starts reading shared memory for DAC values, as written by slave processes. This data is written to DMA memory space, and finally the DAC module is signaled that data is available for transfer.
7) Once per second, the IOP uses the acquired duotone signal, from the TDS timing slave, to calculate the time offset, in μsec, from its cycle 0 ADC data and the GPS 1PPS time mark. It also reads the GPS time from the IRIG-B module.
8) DAQ data is written on every cycle to local memory, as described in Reference 5.
9) Housekeeping primarily has to do with performing various diagnostic checks. These diagnostics are described in Reference 6.

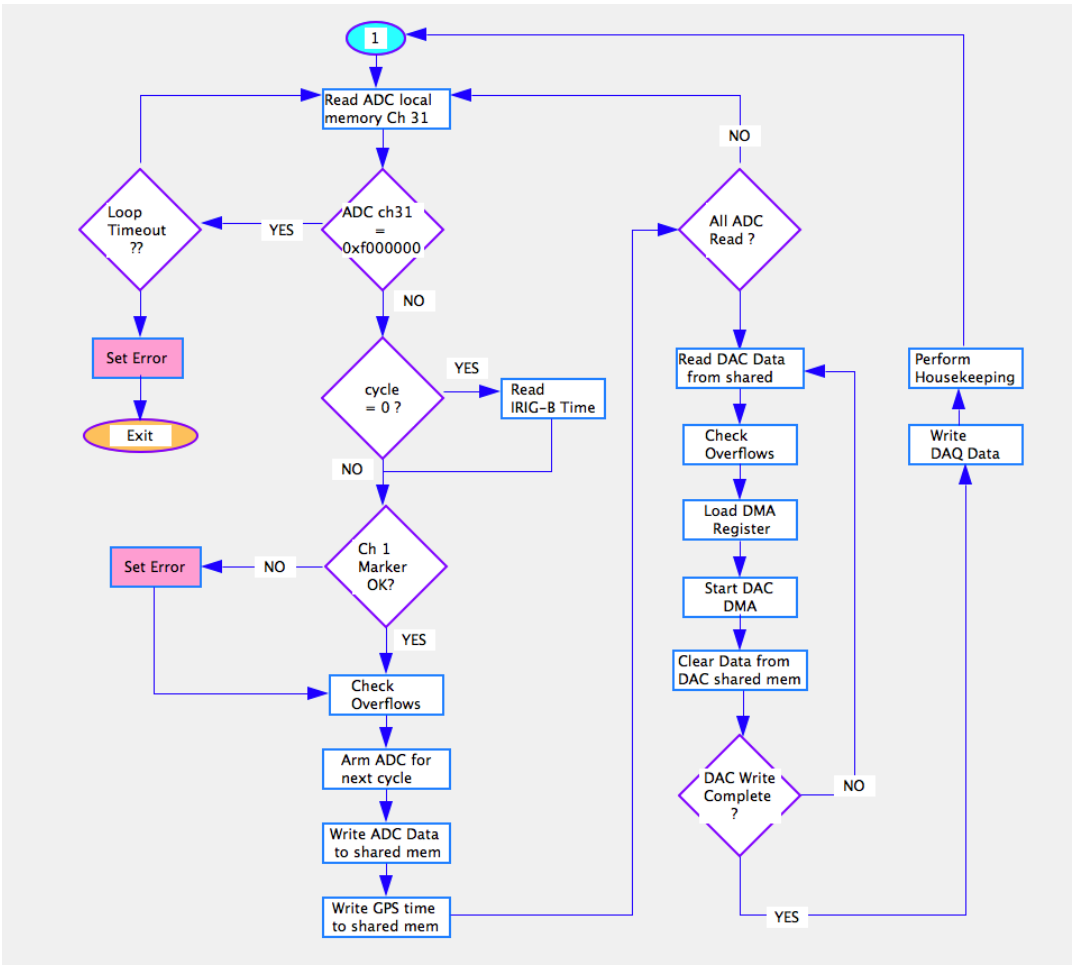


Figure 4: IOP Process Flow Diagram

8 ADC/DAC Shared Memory

8.1 Memory Blocks

Shared memory is established as a circular buffer, with 64 data blocks for each ADC/DAC module. Each data block represents one 65536 Hz sample. Along with ADC/DAC data, these blocks contain GPS second information and cycle count (0-65535) information, for use in marking the data as valid and ready to be read. Both the IOP and slave maintain their own GPS second information and 65536 cycle counters for this verification process.

In the IOP/slave mode, the process sequence for ADC/DAC data read/writes is as follows:

1) IOP (indicated as MASTER in following diagram):
    a. Reads and verifies ADC data.
    b. Writes ADC data to next circular buffer block.
    c. Writes GPS second information and, finally, cycle information.
    d. Reads DAC data from circular buffer block (same cycle count as ADC write)

e. Verifies slave has written correct GPS second and cycle count.
    i. If time/cycle not correct, outputs to DAC module will be a zero.
f. Writes a zero into DAC buffer location (if slave task does not update the value, zero output on next cycle through this buffer location).

2) Slave
a. Detects new, and correct, GPS second and cycle count in ADC data block.
b. Reads data from shared memory and proceeds as normal.
c. Writes its DAC data to the appropriate shared memory block, followed by GPS second and cycle count. The "appropriate" block is always in advance of where the master is reading from and, how far in advance, is dependent on the slave task rate. This is presently hardcoded into the ***controller.c*** source, with the following values write ahead values:
    i. 65K slave: 1 cycle
    ii. 32K slave: 2 cycles
    iii. 16K slave: 4 cycles
    iv. 4K slave: 8 cycles
    v. 2K slave: 16 cycles

Note that for the 2K and 4K slaves, the write ahead is not the expected factor between the 65K IOP and 2K/4K slave. These settings presume that these slave tasks will always complete in less than half their maximum time allotment (488 / 244 μsecond). This lower setting is designed to reduce phase delay.

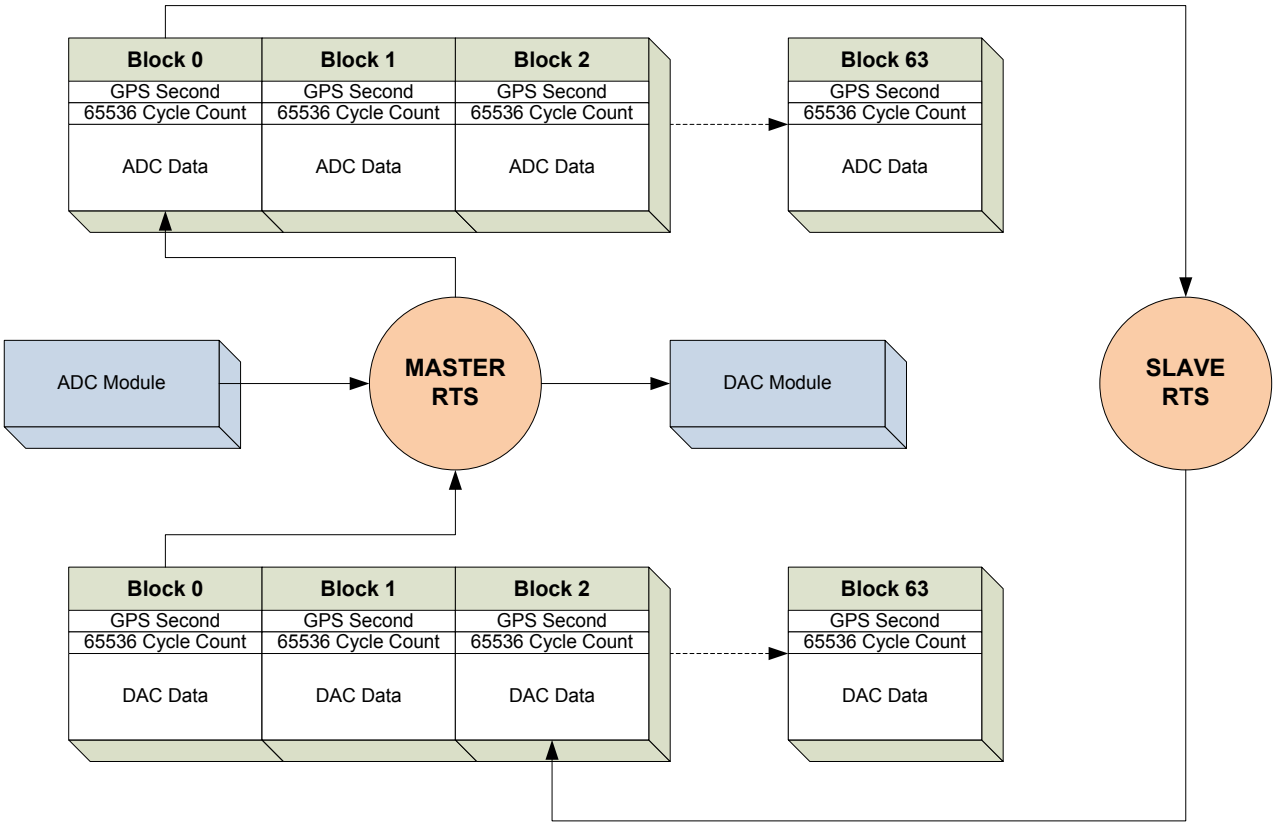


Figure 5: ADC/DAC Shared Memory Blocks

8.2 DAC Module Sharing

The code is designed such that multiple real-time slave processes can access any ADC module, any channel and multiple processes can write to the any DAC module, but not the same DAC channel. ADC module/channel sharing is straight forward, as there is a single writer and multiple readers. DAC module sharing is a bit more complicated.

The IOP process is always started first, as slave processes will not run without it. With only IOP running, the DAC shared memory blocks will not get a new GPS second or cycle count, and therefore the IOP will send out a zero value on all DAC channels to all DAC modules.

Each slave process is compiled to write DAC data only to those channels assigned in the RCG model. However, it will write to the GPS/cycle header for each DAC module it uses. This can result in multiple slaves writing GPS/cycle info to the same DAC block header. So, how is this going to work? This can probably best be explained by example.

8.2.1 Example 1

Two slave processes are defined using the RCG, both using DAC 0, but model 1 using the lower 8 channels and model 2 using the upper 8 channels. The IOP is started and all DAC 0 outputs are constantly set to zero.

1) Model 1 is started:
    a. Model registers to use DAC 0, first 8 channels.
    b. Writes data to first 8 DAC channels, then writes GPS/cycle header.
    c. IOP sees correct GPS/cycle and outputs data to DAC. Since model 1 is not writing data to shared memory for upper 8 channels, these remain at zero, as IOP is constantly resetting the DAC memory blocks with zero values after transmission.
2) Model 2 is started:
    a. Model registers to use DAC 0, last 8 channels.
    b. Writes data to upper 8 channels, then writes GPS/cycle header.
    c. IOP still sees correct GPS/cycle (if both model 1 and 2 are running/writing correctly) and writes out the 16 values provided by both models.

If either model 1 or 2 is later stopped, to load new code, etc., then zeros will be output for that model's associated channels. If both are stopped, then revert to IOP only, which sends zero to all DAC channels.

8.2.2 Example 2

Two slave processes are defined using the RCG, both using DAC 0, but model 1 using the lower 8 channels and model 2 also using the lower 8 channels (by user error). The IOP is started and all DAC 0 outputs are constantly set to zero.

1) Model 1 is started: Same behavior as in Example 1, item 1 above.
2) Model 2 is started:
    a. Model registers to use DAC 0, first 8 channels.
    b. Since model 1 has already registered to use these channels, model 2 exists with channel conflict error message.

This "registration" feature is what prevents multiple slave applications from attempting to write to the same channel of the same DAC module.

9 Slave Operation

Applications which run actual control algorithms are compiled by the RCG in slave mode, as defined by "adcSlave=1" in the code model parameter block. In this mode, the same ***controller.c*** source file is used in the compilation as is used for the IOP.

Once the IOP is started, the slave applications may be started. There may be as many slave applications as there are CPU cores available on the computer, less two (one for Linux non-real-time tasks, and one for the IOP task).

9.1 Initialization

Code initialization is similar to the IOP. The main difference is that, instead of calling map.c functions to find and initialize I/O modules, the slave receives this information via IPC shared memory from the IOP. For ADC and DAC modules, it establishes pointers to the ADC/DAC shared memory buffers instead of ADC/DAC card I/O pointers.

9.2 Startup Synchronization

Once initialization is complete, the slave enters its infinite loop. To synchronize with the IOP, it waits for Block 0 of the first ADC shared memory buffer to contain a cycle count of zero, which only occurs coincident with the ADC sample taken at the GPS one second mark. This triggers the code to enter its infinite control loop.

9.3 Sequencer Runtime

Upon detection of the first ADC read, the sequencer will begin the infinite loop, and remain synchronized with the IOP, as shown in the following figure.

A few items of note:

1) All real-time tasks, regardless of user defined rate, will only take the first ADC sample for its first code cycle on startup. Thereafter, it will read 65536/FE_RATE (where FE_RATE is the user defined 2^n code rate) samples before proceeding to call the user application, etc. This ensures that all CDS code is synchronized to the same time mark.
2) Each sequencer maintains two internal cycle counters, which roll over once per second. These counters are used by the sequencer to schedule code which is not executed on every cycle, such as writing to the DAQ network, and to balance CPU time from cycle to cycle with various housekeeping activities, such as EPICS data transfers, etc. These counters always start at zero, coincident with the 1PPS startup signal.
    a. 0-65535, to track individual ADC read cycles.
    b. 0 to (FE_RATE – 1).
3) Every system, regardless of rate, reads all 65536 samples/second individually. This does not mean that systems running at lower rates have to be ready to accept data synchronously at 65536Hz from the IOP. When the sequencer comes back around to the ADC read portion, it will already see the next sample is ready, process it, and see another sample in the next buffer location. In this fashion, the code actually makes use of normally idle time to “catch up” with the IOP.
4) Since slave processes run at less than the 65536Hz sample rate of the ADC/DAC modules, the slave tasks must perform appropriate decimation and upsample filtering. This is accomplished by running each sample through a predefined IIR filter. While the coefficients for this filter are presently set in the source code, it is intended that definition of these coefficients be moved to the RCG to allow users to provide their own tailored filters for this purpose.
5) For upsample filtering between the application and the DAC modules, two methods are provided in the code:
    a. Zero padding (default): The value calculated by the application to be output to the DAC channel is sent into the filter only once. After that, zeros are loaded. For example, for a slave running at 2K, the output calculated by the application must be run through the filter 32 times and produce 32 DAC outputs. In zero padding, the actual value is only passed to the filter once. Thereafter, zeros are passed to the filter to produce the remaining outputs.
    b. No zero padding. The output value is passed to the IIR filter on every pass.
6) Slave processes communicate with Binary I/O (BIO) hardware directly. BIO cards are read once/second and written to whenever code detects a new value from the user application.

7) Inter-Process Communications (IPC) between slave processes, via shared memory or reflected memory, is done directly by the slave application with code provided in the communications library (Reference 4).

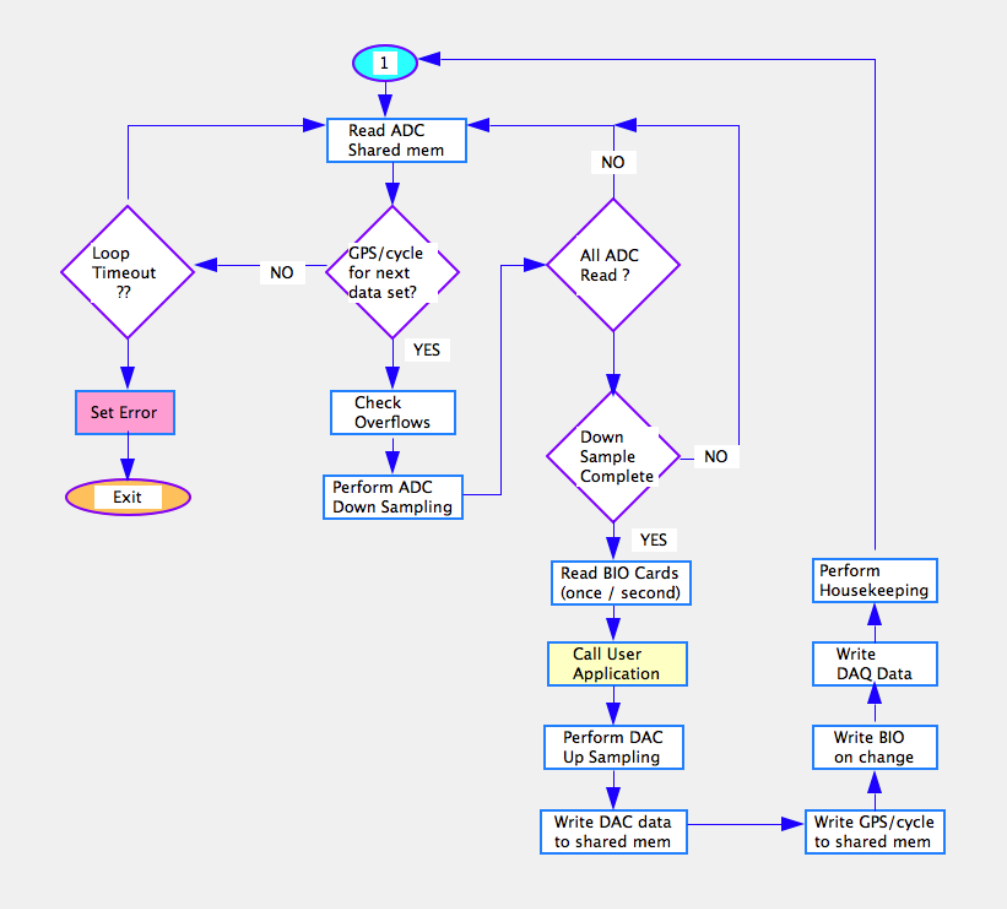


Figure 6: Slave Process Flow Diagram