LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY

LIGO Laboratory / LIGO Scientific Collaboration

LIGO-T0900607-v4 *LIGO* 10/8/2010

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.

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. Separate documentation is yet to be developed (planned). 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. 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.

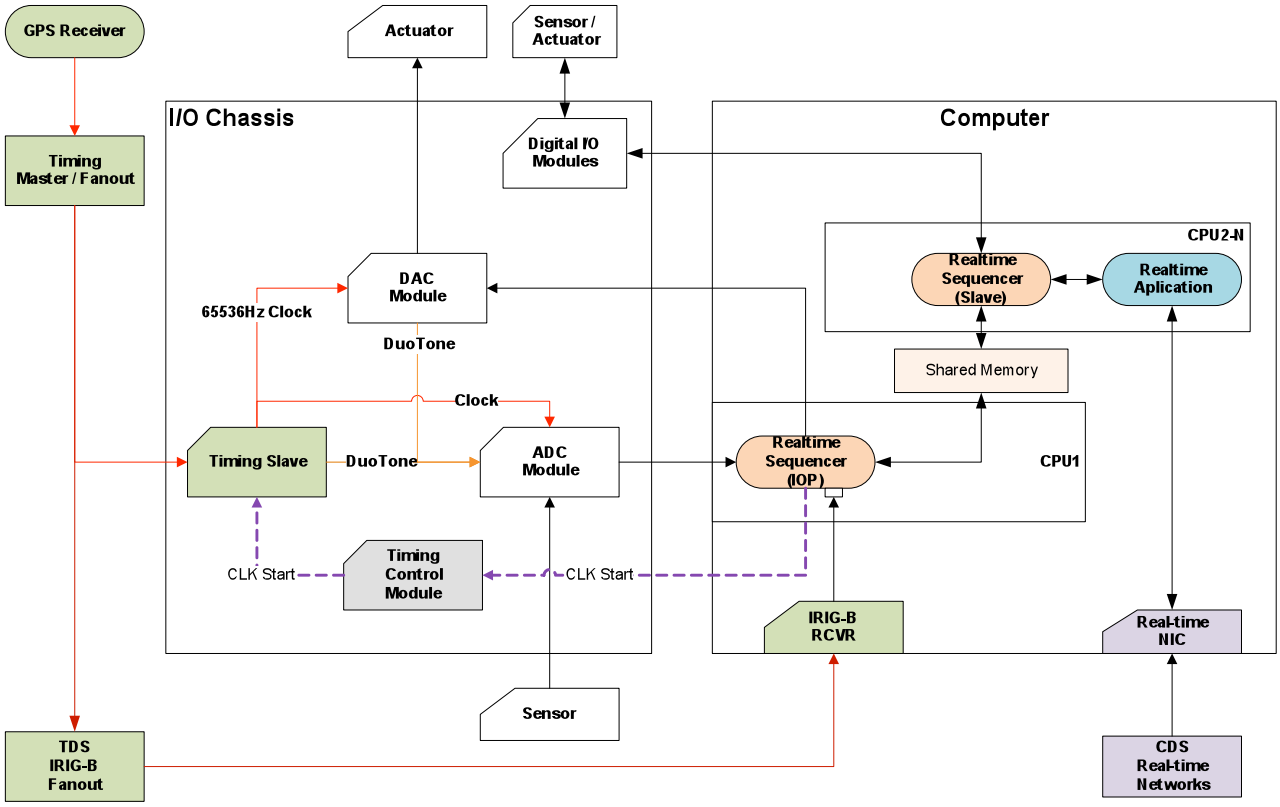


Figure 1: CDS Real-time System Overview

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

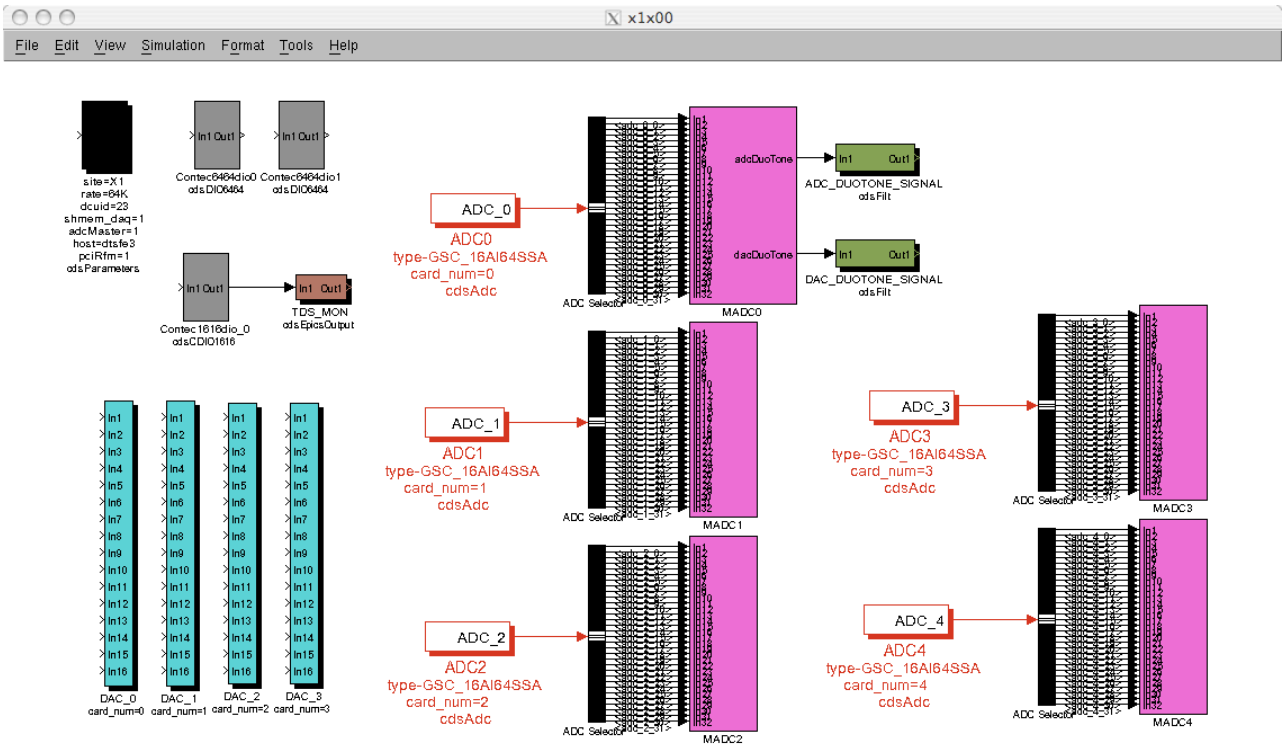
The IOP is defined in a standard Matlab model and compiled by the RCG in the same fashion as any other user real-time application. It is intended that there be a single, generic IOP model, from which all other IOPs are developed. An example is shown in the following two figures, with the first being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

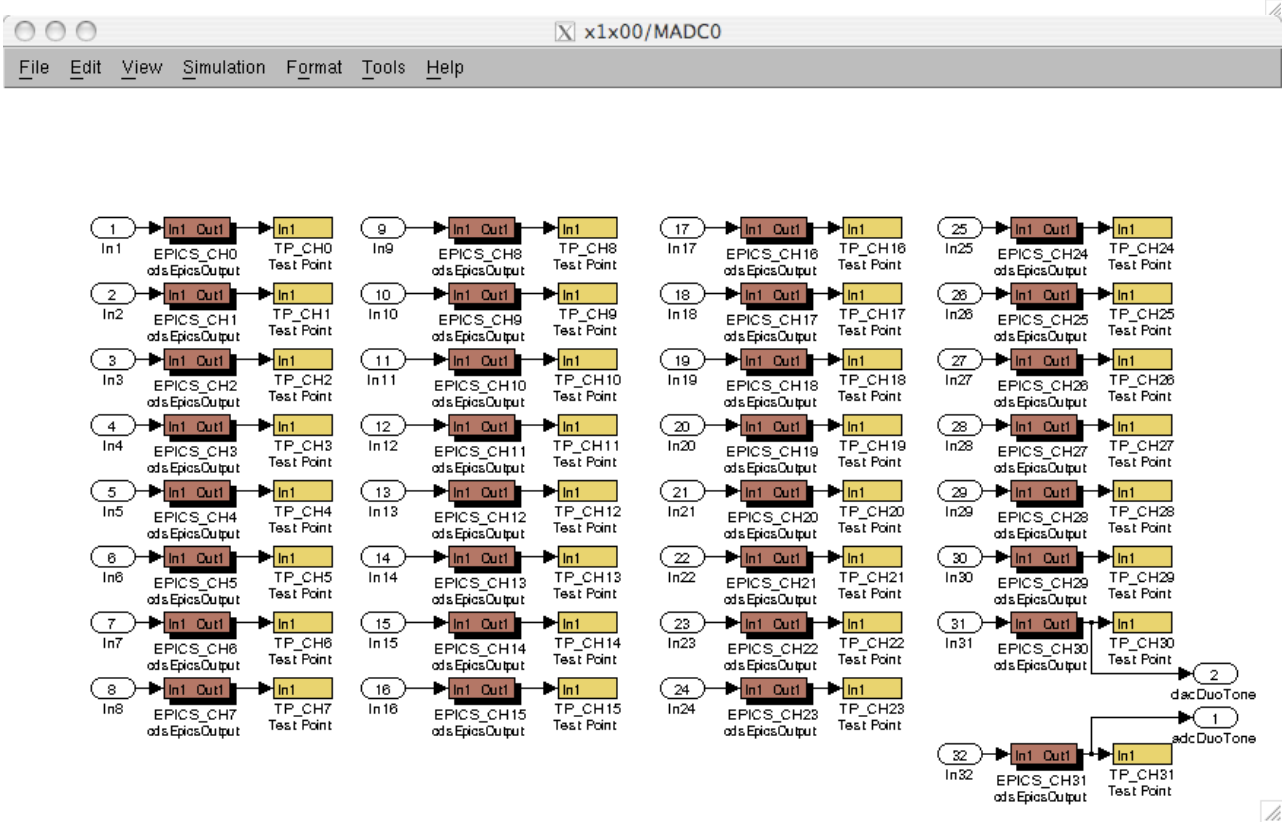
This model almost exclusively contains I/O components. For every I/O module to be installed in the connected I/O chassis, an appropriate I/O part is placed in the IOP model. In this example, there are five ADC modules and 4 DAC modules, along with two binary I/O cards.

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.


x1x00
File Edit View Simulation Format Tools Help
site=X1
rate=64K
dcuid=23
shmem_daq=1
adcMaster=1
host=dtsfe3
pciRfm=1
odsParameters
Contec6464dio0
odsDIO6464
Contec6464dio1
odsDIO6464
Contec1616dio_0
odsCDIO1616
TDS_MON
odsEpicsOutput
ADC_0
ADC0
type-GSC_16AI64SSA
card_num=0
cdsAdc
ADC_1
ADC1
type-GSC_16AI64SSA
card_num=1
cdsAdc
ADC_2
ADC2
type-GSC_16AI64SSA
card_num=2
cdsAdc
ADC_3
ADC3
type-GSC_16AI64SSA
card_num=3
cdsAdc
ADC_4
ADC4
type-GSC_16AI64SSA
card_num=4
cdsAdc
adcDuoTone
dacDuoTone
ADC_DUOTONE_SIGNAL
DAC_DUOTONE_SIGNAL
cdsFilt
MADC0
MADC1
MADC2
MADC3
MADC4
DAC_0
DAC_1
DAC_2
DAC_3
card_num=0
card_num=1
card_num=2
card_num=3


x1x00/MADC0
File Edit View Simulation Format Tools Help
EPICS_CH0
EPICS_CH1
EPICS_CH2
EPICS_CH3
EPICS_CH4
EPICS_CH5
EPICS_CH6
EPICS_CH7
EPICS_CH8
EPICS_CH9
EPICS_CH10
EPICS_CH11
EPICS_CH12
EPICS_CH13
EPICS_CH14
EPICS_CH15
EPICS_CH16
EPICS_CH17
EPICS_CH18
EPICS_CH19
EPICS_CH20
EPICS_CH21
EPICS_CH22
EPICS_CH23
EPICS_CH24
EPICS_CH25
EPICS_CH26
EPICS_CH27
EPICS_CH28
EPICS_CH29
EPICS_CH30
EPICS_CH31
odsEpicsOutput
TP_CH0
TP_CH1
TP_CH2
TP_CH3
TP_CH4
TP_CH5
TP_CH6
TP_CH7
TP_CH8
TP_CH9
TP_CH10
TP_CH11
TP_CH12
TP_CH13
TP_CH14
TP_CH15
TP_CH16
TP_CH17
TP_CH18
TP_CH19
TP_CH20
TP_CH21
TP_CH22
TP_CH23
TP_CH24
TP_CH25
TP_CH26
TP_CH27
TP_CH28
TP_CH29
TP_CH30
TP_CH31
Test Point
dacDuoTone
adcDuoTone

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 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 0x110000 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.

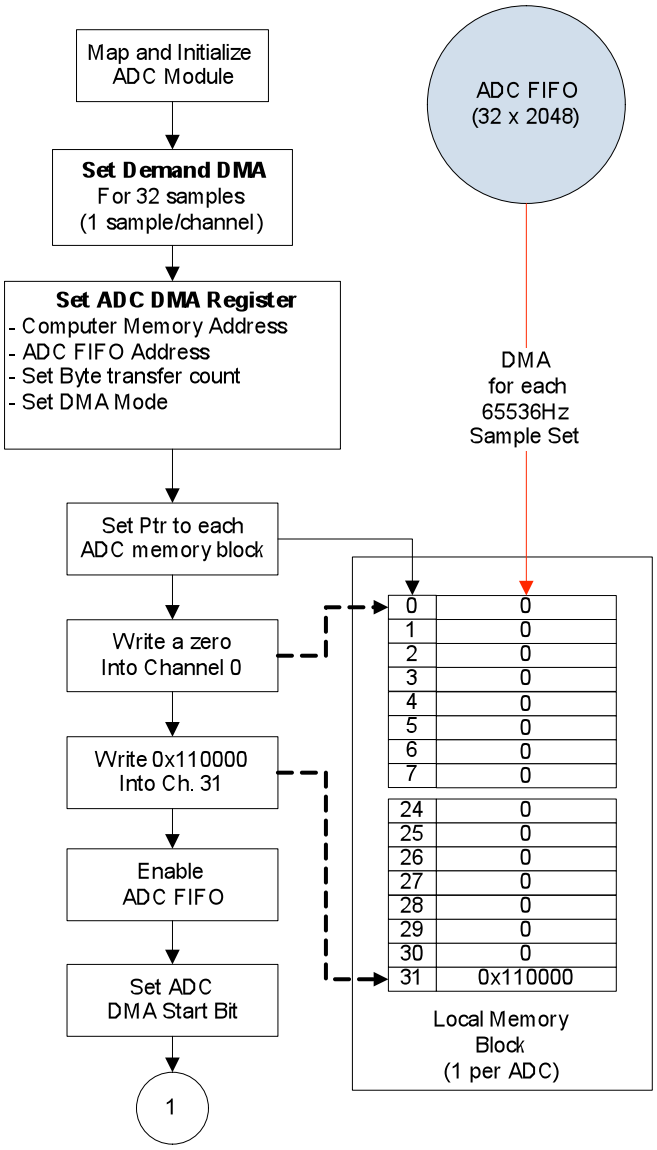


Figure 2: ADC Initialization

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

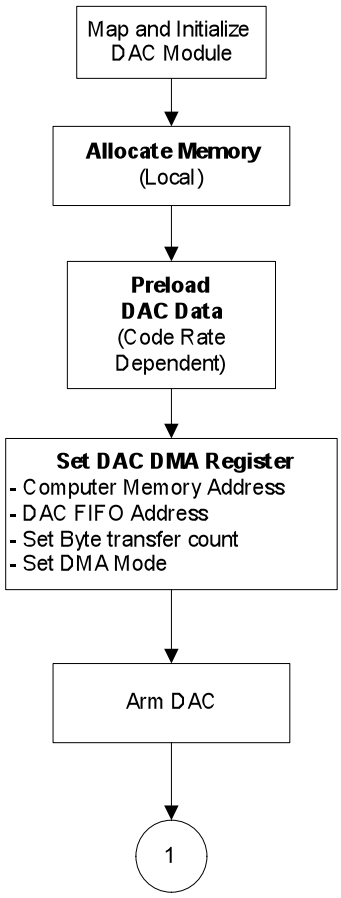


Figure 3: DAC Initialization

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) 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:

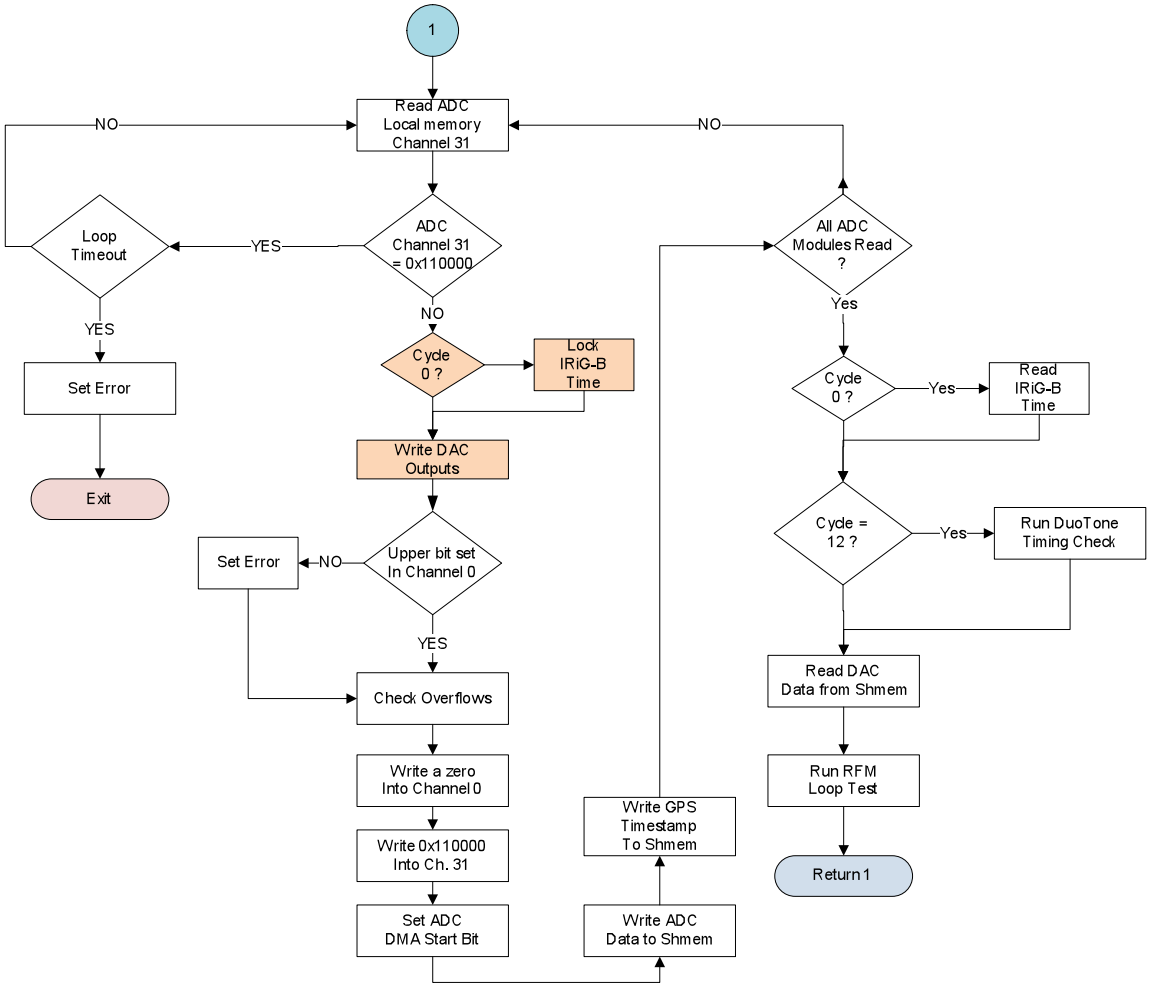
- Read/writes are performed with an unsigned 32 bit integer. The lower 16 bits are read and upper 16 bits are write. Register definitions for the timing control binary I/O module are listed in LIGO-T1000635.
- The first ADC and DAC module interface cards have relays which allow:
    - Duotone from timing 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

7.3.1 Sequence

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 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).
2) After the first ADC module data is available, the IOP performs the additional tasks, as highlighted in orange in the diagram:
    a. If cycle 0, set control bit in IRIG-B module to lock in the time. This is later read out as a timing diagnostic (time offset from 1PPS).
    b. Write all DAC outputs. During the previous cycle, the IOP has gathered up all the DAC outputs from shared memory and prepared them for transfer to the DAC modules. At this point, the IOP triggers the DAC modules to read the data using their DMA engine. This is done right after an ADC trigger to provide the maximum time window for the data to be delivered to the DAC modules prior to the next 65K clock, which causes the DAC output values to change to these new settings.
3) IOP rearms the ADC DMA Start Bit. This enables the ADC module to send data again once it has a sample ready.
4) The IOP writes a GPS timestamp and 65K cycle count along with the ADC data to shared memory. The time and cycle is then inherited by the slave units, after first verifying that this was the expected time and cycle count.
5) 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.



7.3.2 IOP Cycle Timing

Preliminary timing tests have been done on an IOP running on several computers, of the type to be used in aLIGO, with aLIGO IOC and timing system. A more complete and formal test procedure is in development, but initial results are shown in the following figure. For this test, an IOP was run, along with a test slave process, also running at 65KHz, to provide a DAC duotone signal for timing measures. The IOP has a duotone timing test function built in, which was used to measure both the ADC and DAC duotone signals. No external Anti-Alias (AA) or Anti-Image (AI) are connected in the loop. Signals are strictly via the IOC timing interface bus.

Timing readings from all three computer/IOC systems were identical and consistent, from start to restart of the code, and over many hours of continuous operation.

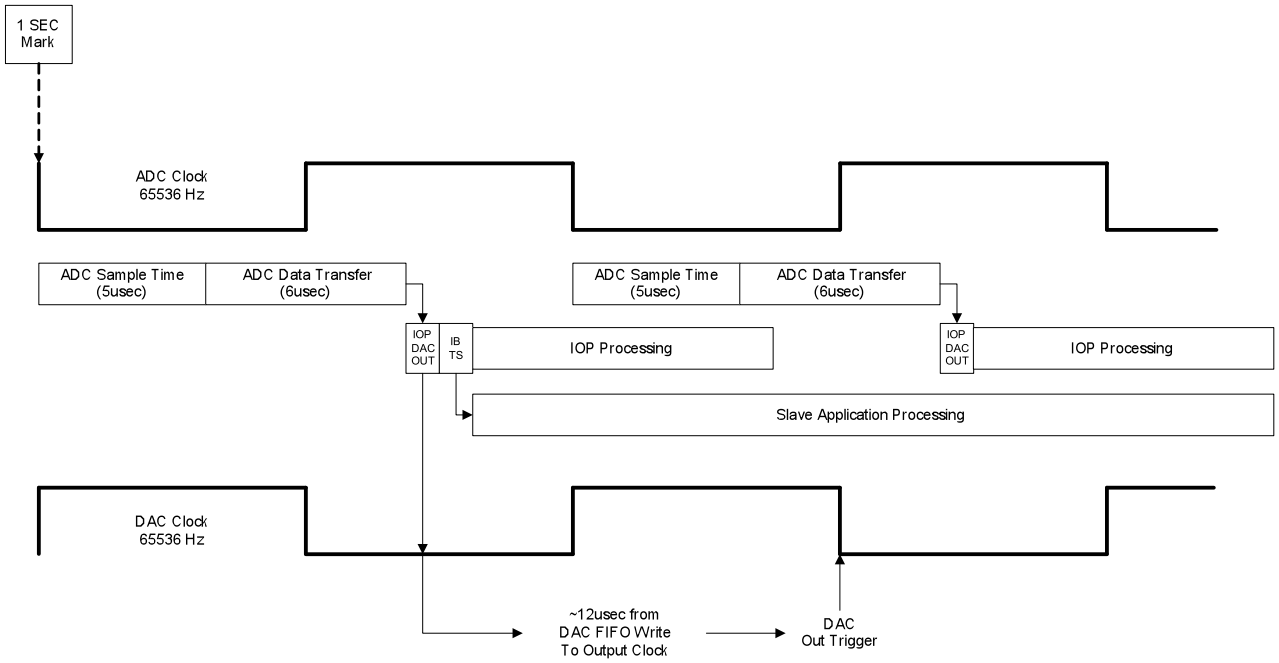
The ADC duotone measurement indicates that the duotone signal crosses zero at +5µseconds after the actual 65536Hz clock coincident with the 1PPS mark (note that ADC inputs and DAC outputs occur on negative clock edges). While this should be further checked with the Matlab based duotone timing verification code, this number is consistent with the duotone to GPS 1PPS signal comparison measurements made with an oscilloscope.

Once the IOP detects new ADC data, it sends a command to the IRIG-B receiver module to lock in the time, in GPS seconds and µseconds. The difference between clock at 1PPS mark and IOP triggering of IRIG-B module is consistently 12 µsecond. Beyond the up to 5 µseconds involved in ADC sampling and conversion (ADC modules rated at 200KSamples/sec), this would indicate an additional delay of 5-7 µseconds in moving the data from the ADC to computer memory (1-2µsecond taken up in polling loop, triggering the DAC module DMA engines to transfer out DAC data, and write to the IRIG-B register). This time only varies by 1-2 µseconds between a computer with 1 ADC and 1 DAC connected and a full complement of a total of 10 ADC/DAC modules.

The start of the slave cycle is delayed an additional 1 to 2 µseconds (max), as the IOP checks and transfers the ADC data to shared memory.

Note that, by default, the DAC clock is selected to be of opposite polarity from the ADC clock. The reason that this is done is to allow the maximum amount of time for DAC data transfers before the next timing system clock causes the DAC to output data on its next negative edge.

Overall processing time of the IOP is typically 6 to 9 µseconds, with maximum DAQ rate of 2MByte/sec set. Further optimization of the code timing, by disabling DAQ, can save an additional 2µseconds. Note this DAQ and other housekeeping occurs after the IOP has transferred ADC/DAC data to/from the shared memory.



8 ADC/DAC Shared Memory

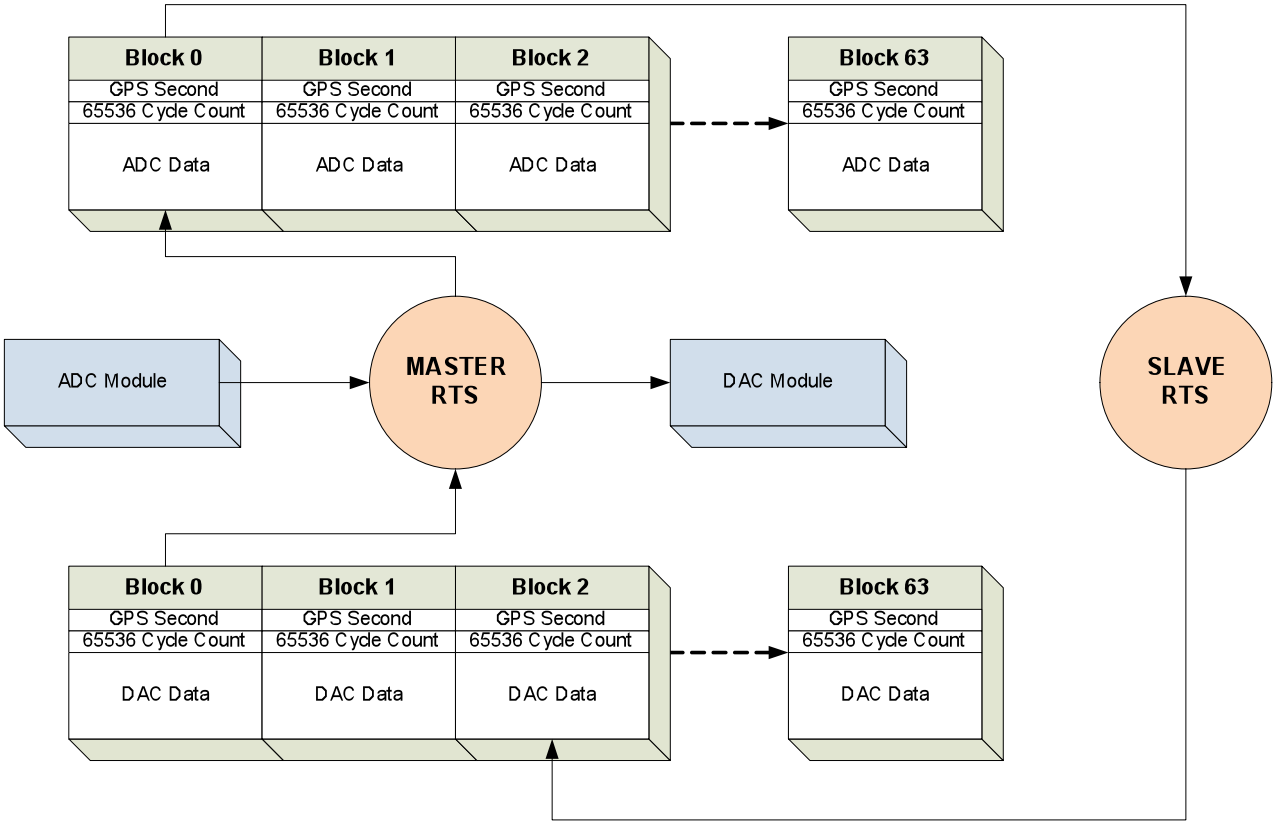
Shared memory is established as a circular buffer, with 64 data blocks for each ADC/DAC module. Each data block represents one 65536 Hz sample. Along with ADC/DAC data, these blocks contain GPS second information and cycle count (0-65535) information, for use in marking the data as

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.



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-realtime 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. The example in the figure depicts a slave process running at 32768Hz.

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. This can be seen in the timing diagram of Figure 3, where the first code cycle only reads sample zero before processing, and thereafter performs 2 reads for each code cycle (in a 32K system).
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 applications write "ahead" to DAC output memory. In the example above, the 32K slave process, on cycle 0, writes its DAC outputs to DAC output memory locations

2 and 3. This is to ensure, as long as the slave runs within its prescribed time, that the IOP will not read data from this block before the data is written by the slave application (race condition).

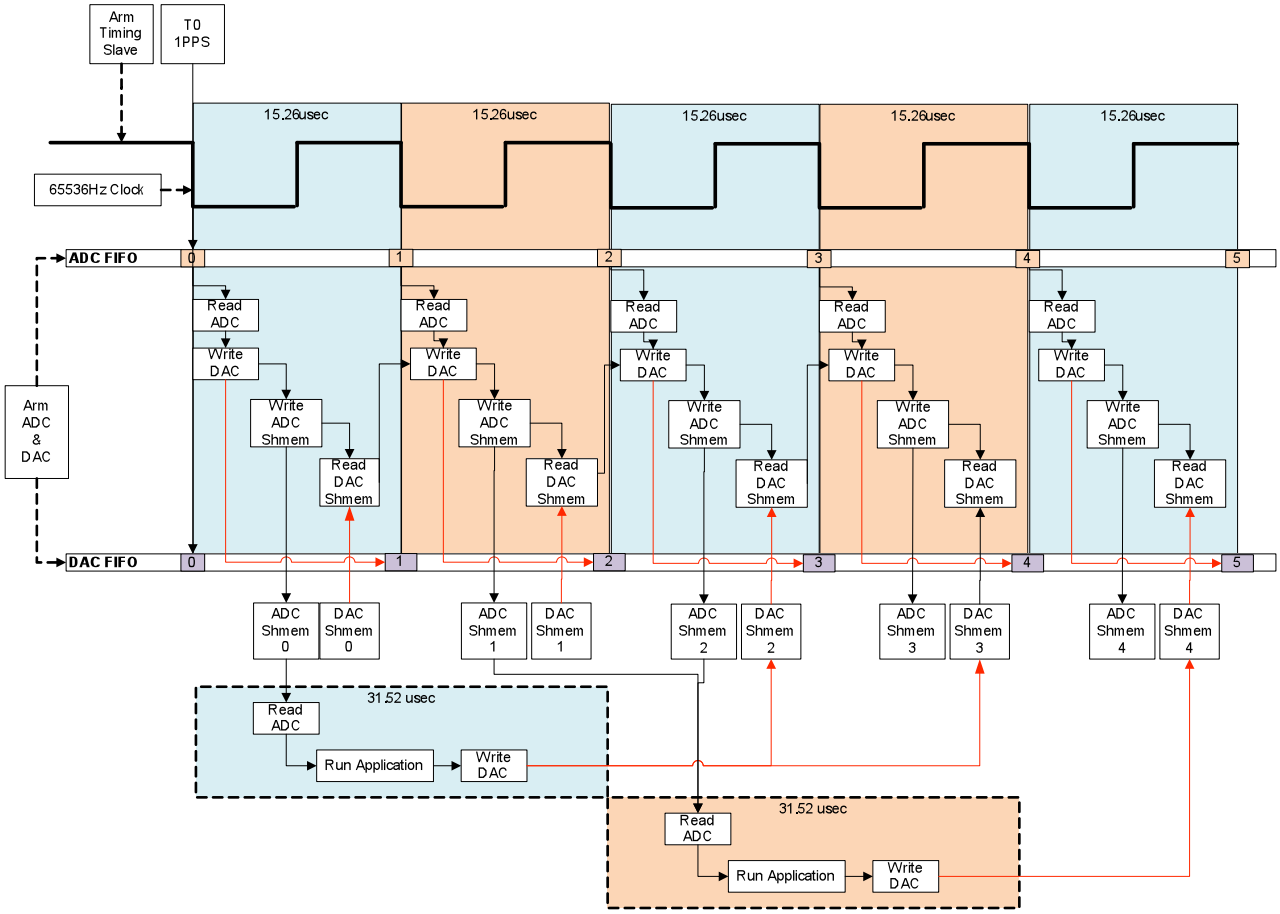


Figure 4: IOP and Slave Timing (Slave at 32768 Samples/sec)

10 Diagnostics

Each RTS provides certain startup and status/error information in a log file located in the application target directory. Kernel I/O driver information may also be accessed via the Linux dmesg command.

A number of diagnostics built into the RTS are reported via EPICS channels for continuous monitoring. Some are in the present code and others are in the process of being added. These include:

1) ADC Timeout: This condition can occur for two reasons: a) Lack of ADC clock or clock at wrong (too slow) rate, or b) the user application run time consistently exceeds

the time allotted for the specified code rate. In this latter case, the ADC FIFO will overflow and the ADC will no longer send data. Is shown in the flow diagram, the code will exit on this condition.

2) ADC FIFO sample count. The ADC modules have a register which indicates how many samples are presently in the FIFO. Ideally, once the sequencer has read the number of samples necessary to begin a new cycle, the FIFO should be empty.
3) The first channel of every ADC read should have an upper bit set. If not, this is an indication of 'channel hopping' or some other timing issue.
4) In a fashion similar to the Matlab duotone timing application developed for testing in ELIGO, the sequencer checks time offset from 1PPS at the beginning of each second. For code performance reasons, this is not as complete as the Matlab version. For example, it presumes that the system started, at worst, within a few clock cycles of the 1PPS mark and only uses the first 12 samples of each second to do the line fit calculation. However, given those caveats, it has shown numbers consistent with the Matlab code in testing. The calculated offset, in μsec, from the 1PPS mark is passed on to an EPICS channel.
5) IRIG-B time offset from 1PPS (in μsec). As soon as the IOP task sees first ADC triggered at 1PPS mark, it sends a command to the IRIG-B interface card to lock in the time. After completion of ADC/DAC data processing, the code reads the time from the IRIG-B card, updating the GPS second count and providing the μsec portion of the readout as a diagnostic.
6) Longest time (in μsec), during a one second period, that the code took to execute one cycle. This is useful in determining if the application is running within the time constraints of its defined rate. Note, however, that the time shown in the CPU_METER EPICS record only includes the time from the ADC read which triggered the cycle until it completes a cycle and is ready to read again. It does not include:
    a. Time it takes for ADC data to transfer from the ADC to local memory. This function is done by the ADC.
    b. Time it takes to transfer data to the DAC modules. For a DAQ write, the sequencer simply writes data to local memory, then sends a DMA start to the DAC. The DAC then becomes the bus master and handles moving the data from computer memory into its FIFO.
7) Longest time (in μsec) for a single code cycle since last 'DIAG RESET' executed by an operator. This code cycle includes time to process all I/O, run the user application part of the task, and perform DAQ and other housekeeping functions.
8) Time it takes to run the user application part of the code (in μsec). Since the IOP does not run a user application, it reports the amount of time it took to process ADC/DAC data and trigger the slave application to run.
9) The code still supports the use of a 1PPS signal into the first ADC as an alternate synchronization method. In this case, the code checks, once per second, that the 1PPS pulse (~1msec in duration) is still in the proper location.
10) DAC FIFO sample count. With the 16bit DAC modules, it is only possible to check if the DAC FIFO is empty. This would be checked prior to a DAC write, and, for systems running at less than 65536, the FIFO should not be empty. A FIFO overflow could also be checked, but, because of the way this is set up, it is not a precise measurement, though would be an indication that something really bad happened (no DAC clock). The

18bit DAC modules, to be used in all suspension systems, does have a DAC FIFO sample count, similar to the ADC module, which would provide more precise information.

11) Real-time network link status. See CDS Inter-Process Communication Software Design LIGO-T1000587 for details.
12) Application DAQ traffic, including channel count, test points selected, and total data rate in KB/Second.
13) ADC/DAC input/output value overflows.