LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY

LIGO Laboratory / LIGO Scientific Collaboration

LIGO-T0900607-v2 *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) CDS Realtime Code Generator (RCG) Application Developer's Guide (T080315)

3 Overview

The LIGO CDS contains of a number of computers which run 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 32768Hz.

In order to facilitate this synchronization, the LIGO Timing Distribution System (TDS) provides a 65536 Hz clock to all real-time systems. This clock is locked to the Global Positioning System (GPS), thereby providing the same timing to multiple LIGO systems and sites. A simple block diagram of the TDS and connections to the real-time computers and processes is shown below. For more information on the TDS, see Reference 1. The primary focus of this document will be on the "Realtime Sequencer" software, highlighted in orange in the diagram.

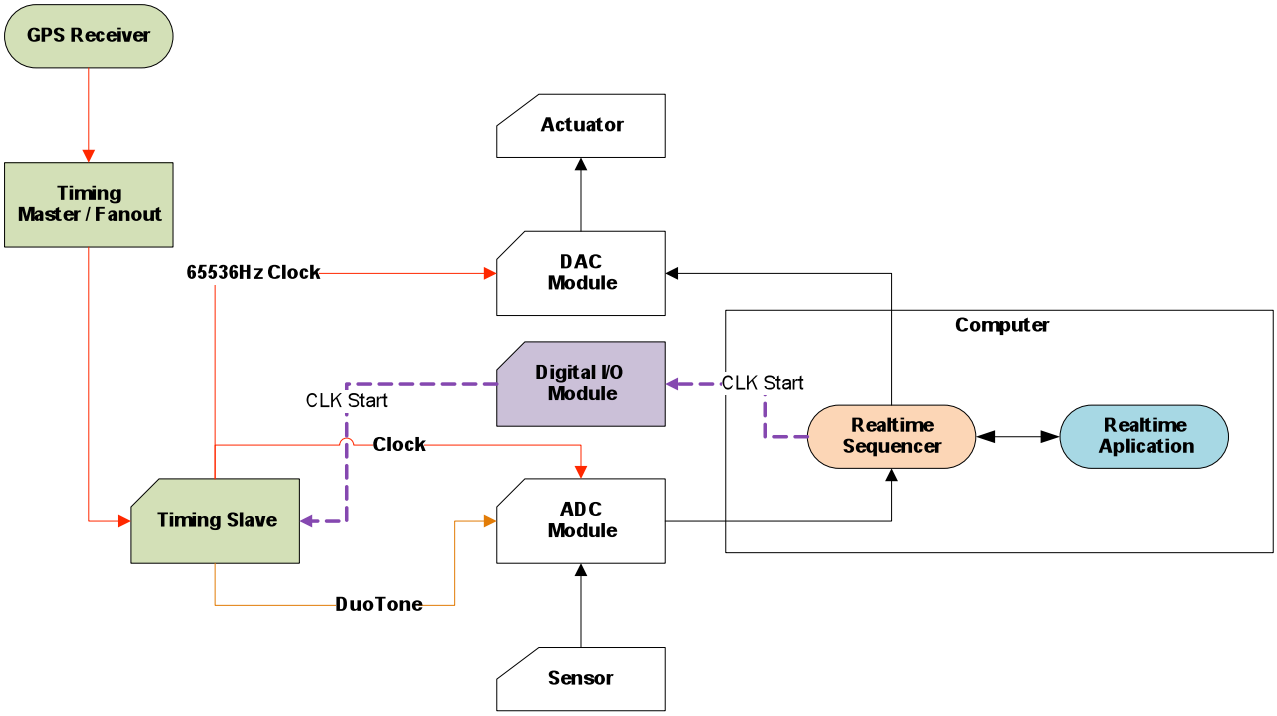


Figure 1: CDS Timing Overview

While the individual application software will vary from computer to computer, the real-time software design calls for a standard set of core software to be used in all real-time tasks. This core software consists of two primary components:

1) Real-time Sequencer (RTS): This code is intended to provide the following functions:
   a. Interface with all aLIGO supported Input/Output (I/O) modules, including initialization, data transfer and error detection capabilities.
   b. Provide for operation of the real-time application in synchronization with the aLIGO timing system and provide appropriate timing diagnostics.
   c. Transfer of data between the real-time code and EPICS.
2) DAQ: Scheduled by the real-time sequencer, this code provides DAQ and global diagnostic data interfaces between the real-time application and the DAQ system.

The focus of this document is the design of the RTS. Information on the real-time DAQ software can be found in LIGO-T0900638.

4 RTS Software Requirements

The primary functions of the RTS are to provide a real-time "scheduler" and an I/O interface for one or more real-time applications running on a CDS control computer. In this capacity, the RTS must meet the following requirements, outlined in detail in CDS Real-time Control Software Requirements, LIGO-T0900603:

1) Support the mapping, initialization and data transfer functions for all CDS standard I/O modules.

2) Synchronize all real-time code operations with the timing clocks provided by the aLIGO Timing Distribution System (TDS).
3) Support application code rates of 2048, 4096, 16384, 32768 and 65536 samples/sec.
4) All clocks from the TDS to the various ADC and DAC modules run at 65536Hz, with an ADC/DAC sample at each clock. Therefore, the RTS must provide appropriate down sampling and interpolation filters to match the application code rate with the I/O clocking rate.
5) Provide accurate GPS timestamp information for DAQ and other real-time data tagging functions.
6) Provide for the exchange of data with other tasks, running within the same computer, via shared memory.
7) In support of the real-time applications, relay data to/from EPICS:
    a. Filter module coefficients
    b. All filter module setpoints/readouts
8) It is intended that the RTS be compiled as the core of every real-time application. For flexibility of use, the RTS shall have two compile options:
    a. MASTER: The RTS directly handles all, or a user defined subset, of the I/O connected to a real-time control computer. In this mode, the RTS must send/receive I/O data to/from other user applications running on separate CPU cores within the same computer. It must also provide synchronization and I/O diagnostic information to the other real-time tasks.
    b. SLAVE: The RTS communicates I/O data via an RTS MASTER.
9) Provide runtime diagnostics via EPICS channels. This is to include:
    a. ADC diagnostics
        i. FIFO overflow
        ii. Timeout (ADC data did not arrive in the specified sample time, indicating an ADC clocking problem).
        iii. Proper channel ordering (first channel of each ADC is tagged in the data by the ADC).
        iv. Individual channel value overflow counters (>=32768 or <=-32768 counts).
    b. DAC diagnostics:
        i. FIFO empty. To ensure a consistent DAC delay, the RTS must always write data to the DAC modules before the DAC FIFO is allowed to run empty (except for systems running at the highest supported rate of 65536Hz).
        ii. FIFO full: DAC is not being clocked properly.
    c. Timing Diagnostics
        i. Duotone comparison. The TDS provides a 960/961Hz duotone signal, which may be connected to ADC channels. The RTS shall provide an algorithm, using this signal, to check its absolute timing in respect to the GPS 1PPS mark. This algorithm shall run once per second and report the result to an EPICS channel as a variation from 1PPS in microseconds. In addition, this signal shall be written as a DAQ channel for additional checking by DMT and other timing monitor software.

ii. The design of the CDS I/O chassis allows for direct connection of certain DAC channels back to ADC channels. The RTS shall make use of this capability to route the incoming duotone signal out to a DAC channel, then read back the data from the connected ADC channel. Using the same algorithm as above, the RTS shall report the resulting delay.
iii. The design of aLIGO CDS includes a "Time Master", which transmits GPS time, in seconds and nanoseconds, to the real-time control network at a rate of 65536Hz. The RTS shall read this timestamp and perform comparisons with its own time and report any variations. The RTS shall also write its timestamp and cycle count to the real-time control network. The latter serves two purposes:
    1. Allows checking of all computer timing by the "Time Master" process.
    2. Allows for real-time data network connectivity checking.

5 RTS Software Design Overview

To accommodate synchronous and deterministic operation, a set of code, common to all such applications, has been developed for LIGO CDS. This code is part of the CDS Realtime Code Generator (RCG), as described in Reference 2.

This code is designed to run with a Linux operating system.

5.1 Real-time Operation and Synchronization

Traditionally, real-time systems employ a common real-time, preemptive scheduler, provided by the operating system. The code itself then sets priorities, interrupts and semaphores to trigger operation of various code threads. In CDS, real-time tasks do not use this standard method. Instead, the CDS code:

1) Has a standard sequencer code thread, the focus of this document, inline compiled with every user application.
2) When run, this executable code (compiled as a kernel module), locks itself onto a single CPU core within a multi-core computer. In doing this, this core is actually removed from the standard Linux scheduler list. This prevents the Linux scheduler from assigning any other tasks to this core or interrupting this core in any other fashion.
3) After various initialization routines, the code uses an assigned Analog to Digital Convertor (ADC) module as its 'scheduler' ie whenever data is received from the ADC, the sequencer begins processing.
4) Once the sequencer has received the proper number of ADC samples for the specified operational rate, it then executes one cycle of code.
5) After running through the sequencer code once, it again returns to wait for the next ADC sample.

Synchronization and time deterministic operation is thereby achieved by having all real-time tasks slaved to ADC modules, which are in turn clocked from a common TDS.

5.2 Operational Modes

The RTS is designed to operate in three modes:

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.

While the 'stand-alone' mode is still supported, as described in Section 6 and 7, the primary configuration for aLIGO controls will be a single IOP and multiple Slaves per real-time computer, as described in Section 8. Some portions of Section 6 and 7 apply to all configurations, so it was opted to leave those sections included in this document update.

The basic flow diagram for the sequencer and representative timing diagram are shown in the following figures. These diagrams, and the description which follows in Section 6 and 7, are for an RTS compiled in "Stand-alone" mode, which is overall representative of all modes.

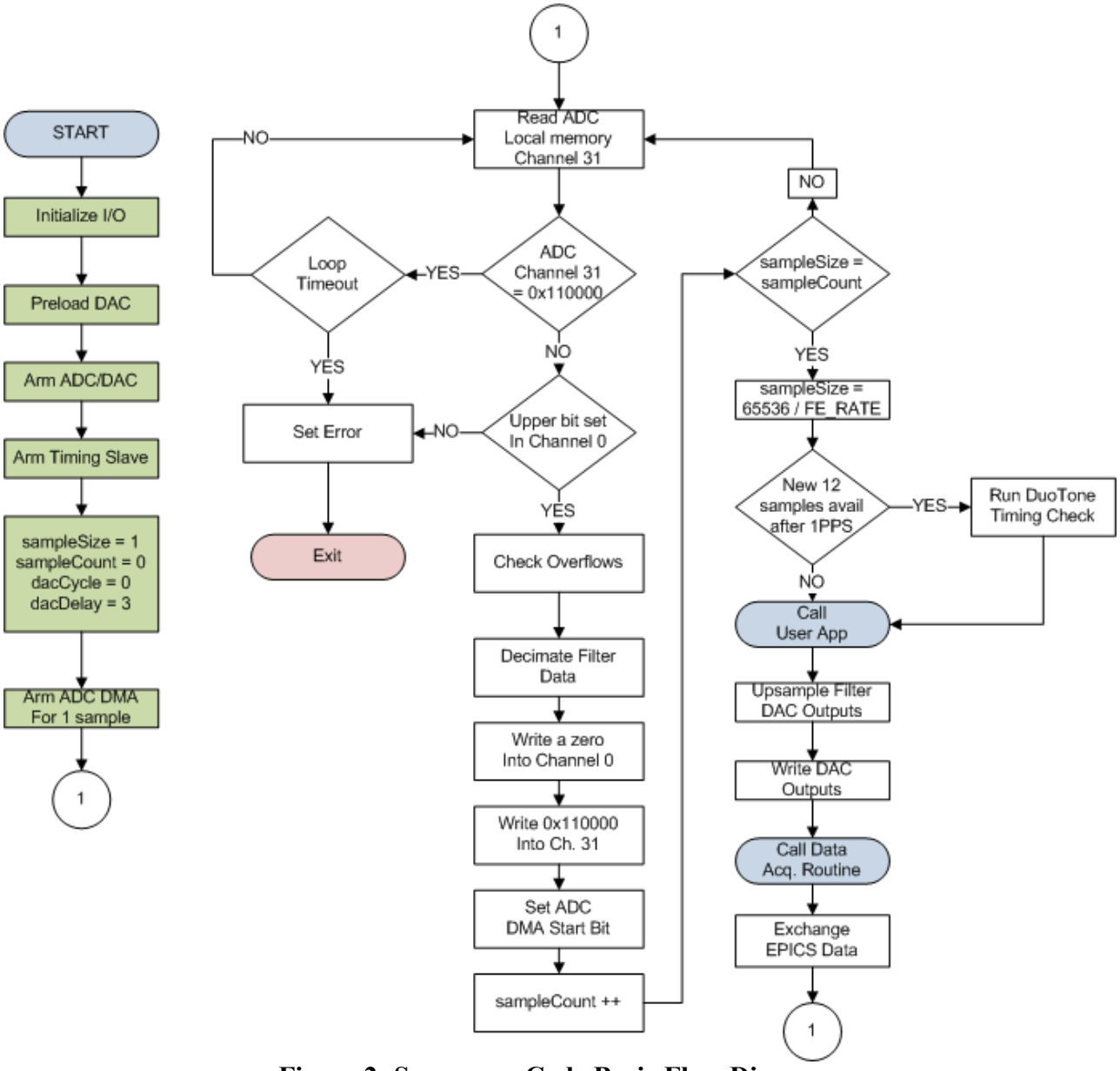


Figure 2: Sequencer Code Basic Flow Diagram

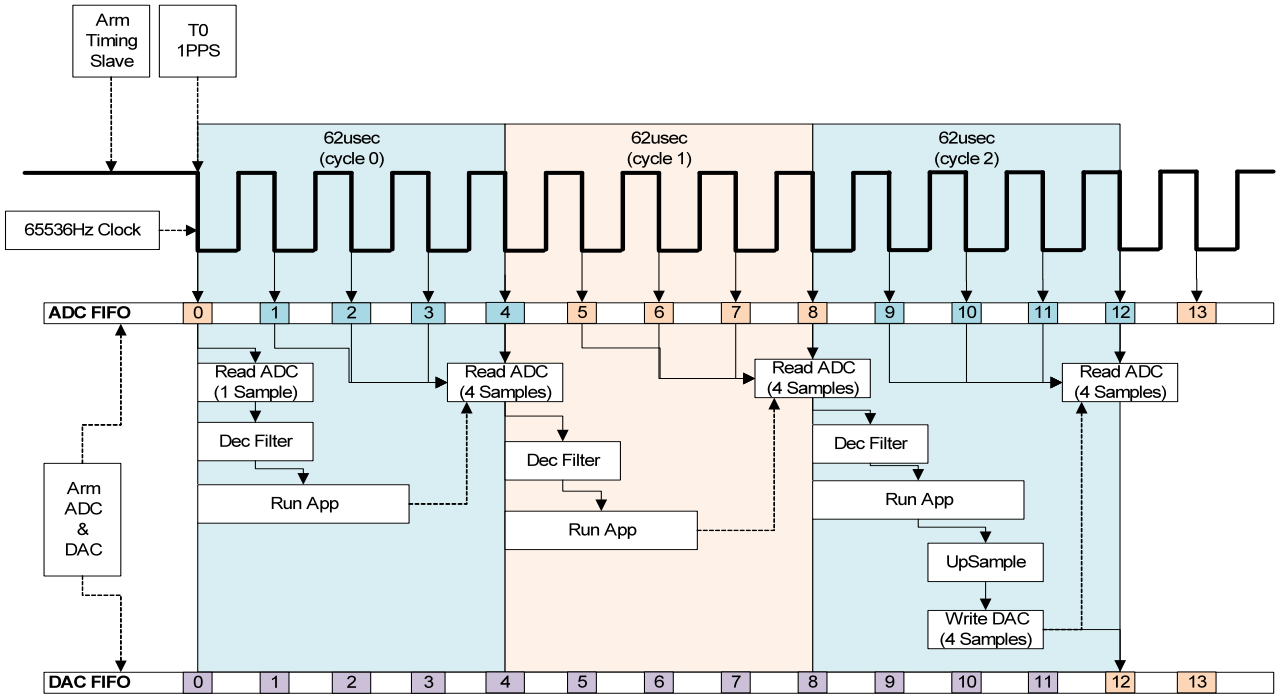


Figure 3: 16384Hz System Timing

6 Sequencer Initialization

Prior to entering an infinite loop, the sequencer must perform various initialization tasks, some of which are highlighted in green in Figure 2. Further details follow.

One of the first steps in initialization is finding, mapping and initializing I/O hardware modules, as defined by the user application. All of the software routines written to provide this initialization and later reading/writing data from/to this hardware are included in the ***cds/advLigo/src/fe/map.c*** file.

6.1 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 an 0x110000 into the channel 31 location. If operating properly, the ADC will never write these values to these locations ie channel zero should 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.

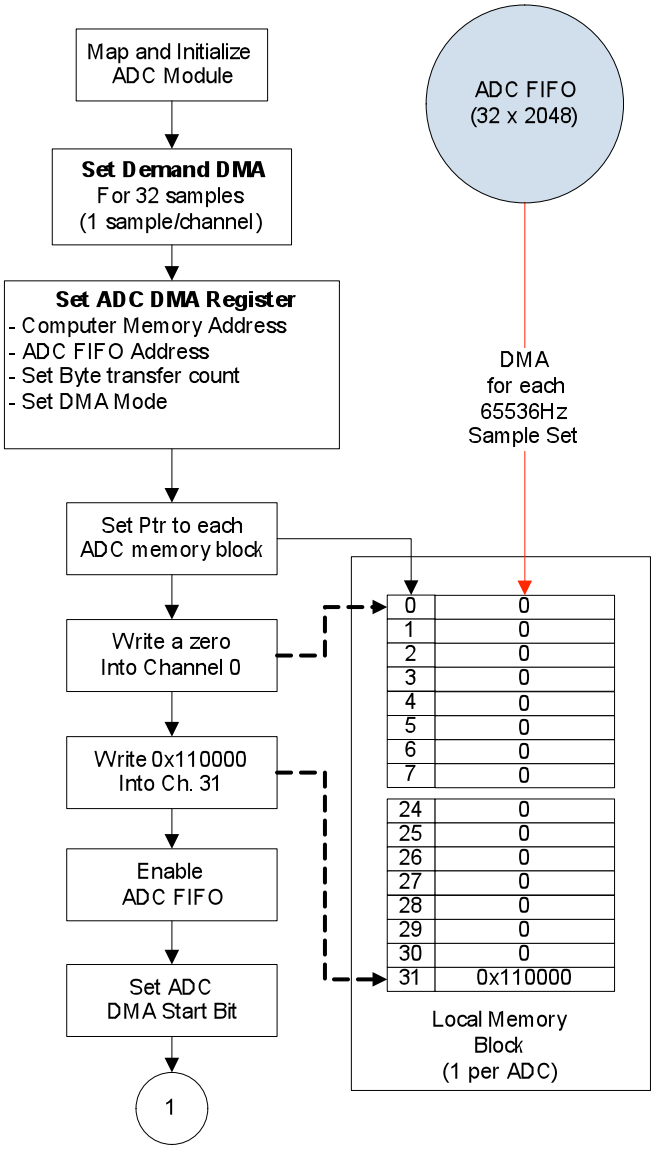


Figure 4: ADC Initialization

6.2 DAC Initialization

Once the ADC modules have been initialized, then the DAC modules are initialized, as shown in the following flow diagram. The primary item of note is the 'Preload DAC Data' block. The purpose of preloading is to ensure that data written to the DAC on any particular code cycle will be added to the end of the DAC FIFO before the FIFO is empty. This is done to avoid DAC output jitter. The time a task takes from ADC ready to DAC write is dependent on the complexity of the application, and may also vary from cycle to cycle by a few microseconds. If the FIFO is allowed to empty before the task is ready to write, tasks may end up writing prior to one 65KHz clock on one cycle, then on the other side of the 65KHz clock cycle the next, introducing jitter noise.

The number of samples to preload is code rate dependent.

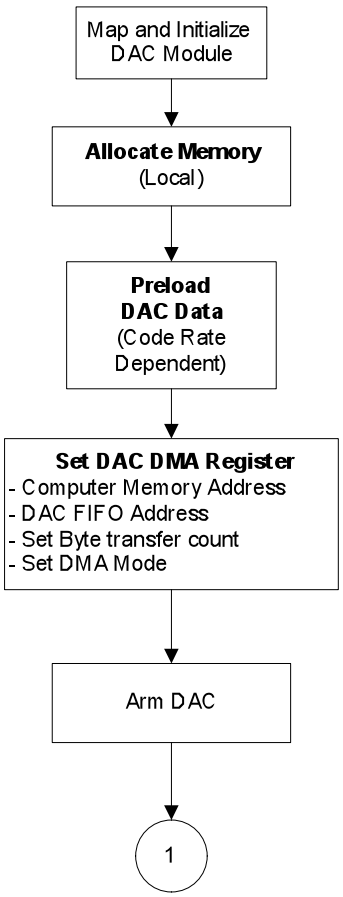


Figure 5: DAC Initialization

7 Sequencer Runtime

After ADC and DAC initialization, the code is almost ready to go into its infinite loop, as outlined in Figure 2. The remaining item is to enable the TDS timing slave. This is done by setting a bit in a PCIe digital output module, which is, in turn connected to the timing slave enable. Once this is set, the timing slave will begin producing 65536Hz clocks coincident with the next 1PPS time marker, which will trigger the ADC and DAC modules to start inputting/outputting data. Upon detection of the first ADC read, the sequencer will begin the infinite loop, at point 1 of the flow diagram shown in Figure 2.

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 4 reads for each code cycle (in a 16K 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). Referring back to Figure 3, the three 62usec blocks shown would be cycles 0, 1 and 2.
3) As previously mentioned, the ADC will automatically send one sample for each of its 32 channels to the CPU local memory whenever it has 32 channels by 1 sample each in its FIFO and the ADC DMA Start Bit has been set. To determine if a new sample is available, the sequencer continuously polls the channel 31 data location until the value has changed from the invalid data that the sequencer previously wrote to that location. Since data arrives in order from channel 0 through 31, this also indicates that the ADC data transfer is complete. After processing the sample data, the sequencer resets the data in the memory block and rearms the ADC DMA to send the next data set when it is ready.

4) 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 ADC. In Figure 3, it shows that the ADC DMA Start Bit is set after ADC data processing, but the code may now go off and call the user application to run, etc. This will typically take longer, often much longer for slow rate systems, than the 15usec before the next ADC sample is written to memory by the ADC. In this case, the ADC will buffer up data in its FIFO. When the sequencer comes back around to the ADC read portion, it will already see the next sample is ready, process it, reset the DMA start bit, and immediately see another sample ready. In this fashion, the code actually makes use of normally idle time to catch up with the ADC.

5) For the first few code cycles after startup, the sequencer does not write values to the DAC

8 Master and Slave Operation

Along with the operating mode previously described, considered the 'stand-alone' mode, the RTS has a compile option to run as a master (I/O Processor (IOP) or a slave. If it is compiled as a master, the RTS handles all defined ADC and DAC I/O and shares out I/O data via shared memory with slave RTS tasks. In slave mode, the RTS redirects I/O from actual hardware to the master shared memory locations. This add a few capabilities:

1) Allows sharing of ADC signals among multiple real-time applications. These applications can read data from the same ADC module and/or the same ADC channels. This can be particularly useful where applications are more compute intensive than I/O intensive.
2) As with the ADC, share DAC modules eg application 1 can write to channels 1-4 of a DAC modules, with application 2 writing to channels 5-8 of the same module.

Operationally, the only major difference between the stand-alone mode and the master/slave configuration is that I/O data is now communicated via computer shared memory. A basic description of how this is implemented is shown in the following figure.

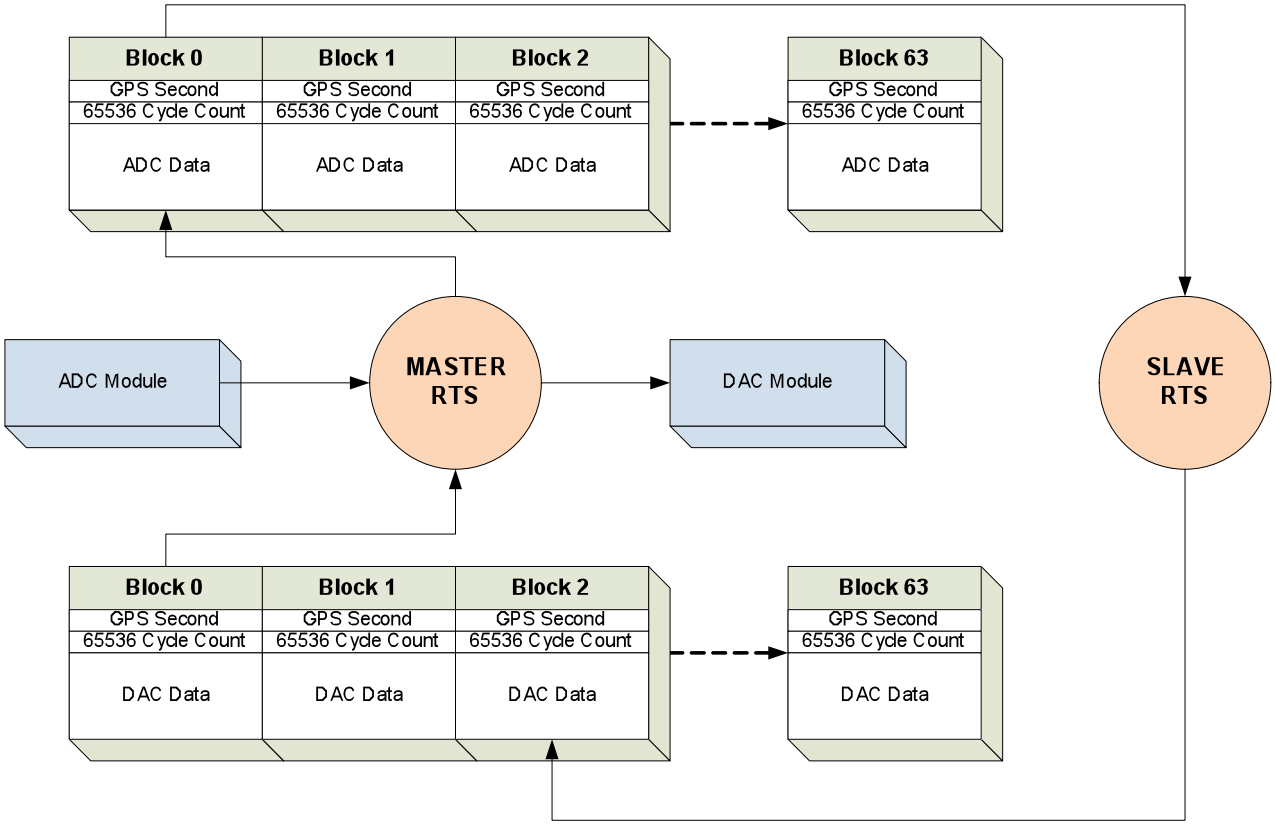


Figure 6: Master/Slave Operation

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 master and slave maintain their own GPS second information and 65536 cycle counters for this verification process.

In the master/slave mode, the process sequence for ADC data reads is as follows:

1) Master (Note: Master must be => highest rate application on the computer):
    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 repeat of the last verified values.
        ii. If time/cycle tag not correct for 64 cycles, master will send zero values to the DAC module.
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

8.1 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.

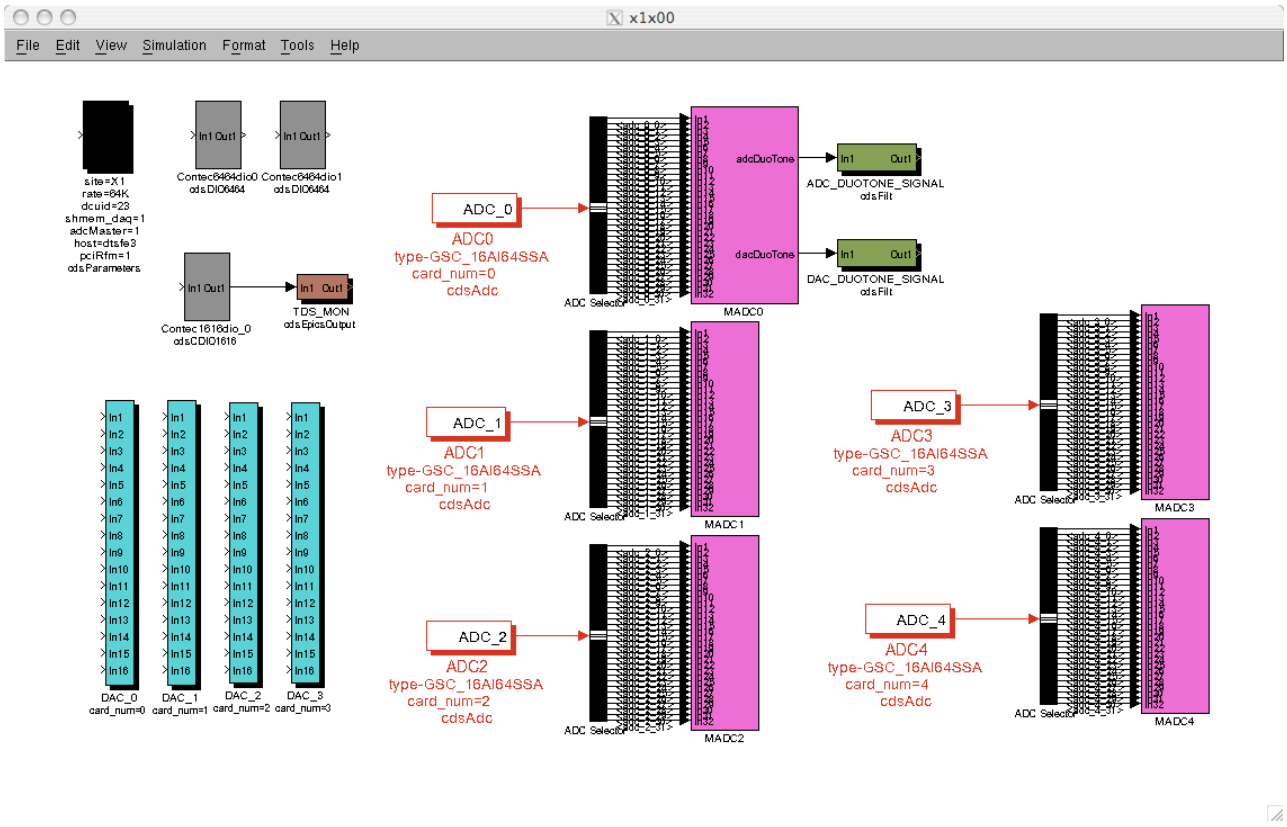
For each ADC module, there are EPICS channels and GDS Test Points defined, which are provided as diagnostics. There are also two filter modules attached to the first ADC, last two channels. These are provided for data acquisition of the ADC and DAC duotone channels, which provide timing diagnostics.

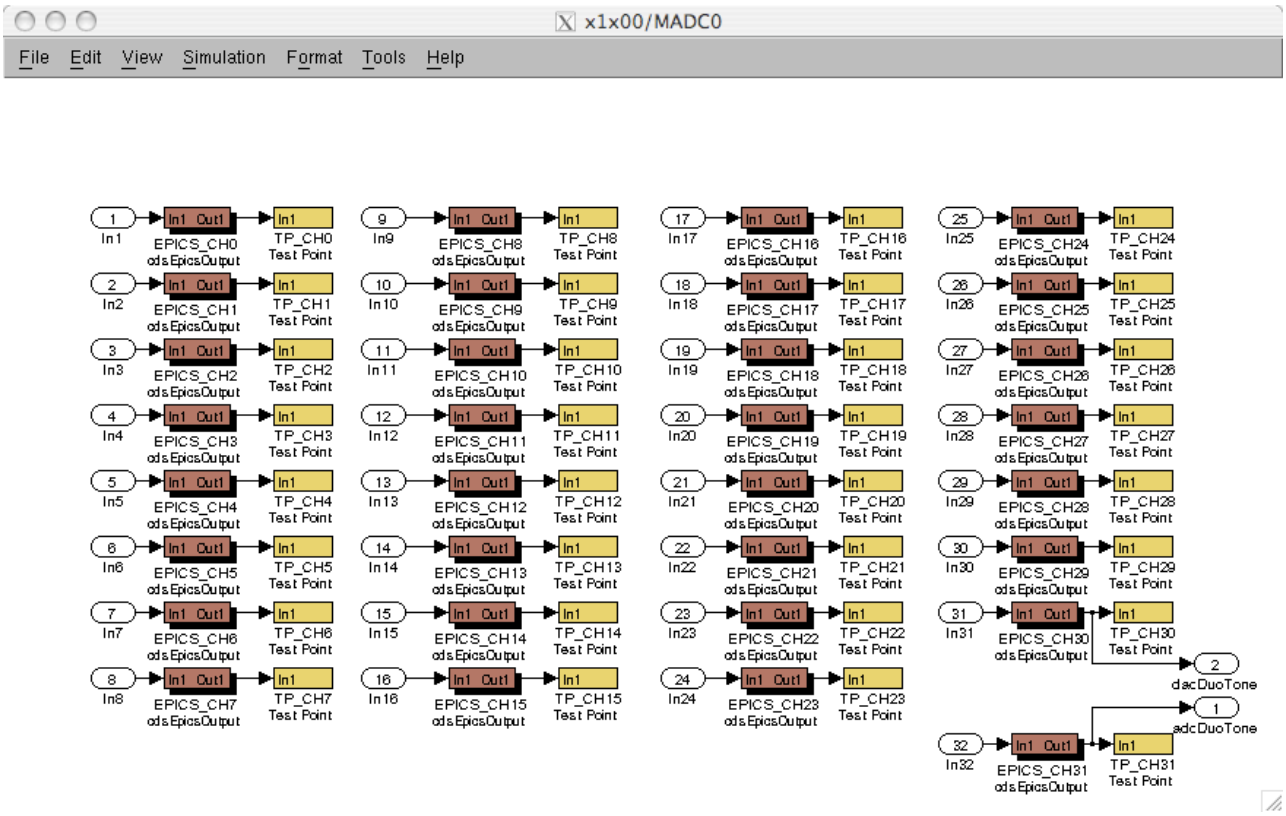
There are also two other significant components in this model:

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 standard, 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.





8.2 IOP and Slave Timing

The following figure shows the timing sequence of an IOP and a single slave (example slave running at 32768 samples/sec).

For an IOP, the initialization is essentially the same as that previously described for a "stand-alone" application. A few differences:

1) The IOP must set up shared memory areas for each ADC/DAC module found on the I/O bus for communication of data with the slave applications.
2) The IOP must pass pointers to this memory to slave applications.
3) The IOP only directly transfers ADC/DAC data between the slaves and the actual hardware ie Slaves communicate directly with other PCIe hardware, and therefore the IOP must pass base address pointers for these devices:
    a. Binary I/O modules
    b. Reflected Memory (RFM) modules

The primary reason that the IOP does not directly handle I/O listed in 3) above is time. These modules can require several microseconds per interaction. Given the 65K performance of the IOP, this can be both a significant additional load and a source of timing jitter for the IOP.

Once triggered to run, the IOP runs through a fairly simple loop, as shown in the following timing diagram, and further detailed in the next section of this document. Some timing numbers for this loop from initial testing:

1) Time to read all ADC modules and write all DAC modules (in full configuration of total combined ADC/DAC count of 10 modules) and pass ADC data to shared memory: <4µsec, typically <2µsec. It is important to keep this time to a minimum, as it cuts into the slave application run time allotment.
2) Total cycle time (ADC trigger, through processing loop, and back to wait next ADC trigger):
    a. <10µsec (typical 7µsec) if running with DAQ turned on. IOP, as with all applications, have compile option to turn DAQ off.
    b. <8µsec (typical 5µsec), with DAQ turned off.

Reduced times are naturally achieved with reduction of I/O modules. This testing was done using a standard CDS I/O chassis connected to both quad core Intel 3.0GHz processor computers and six core Intel 3.3GHz processor computers, with no notable difference in times.

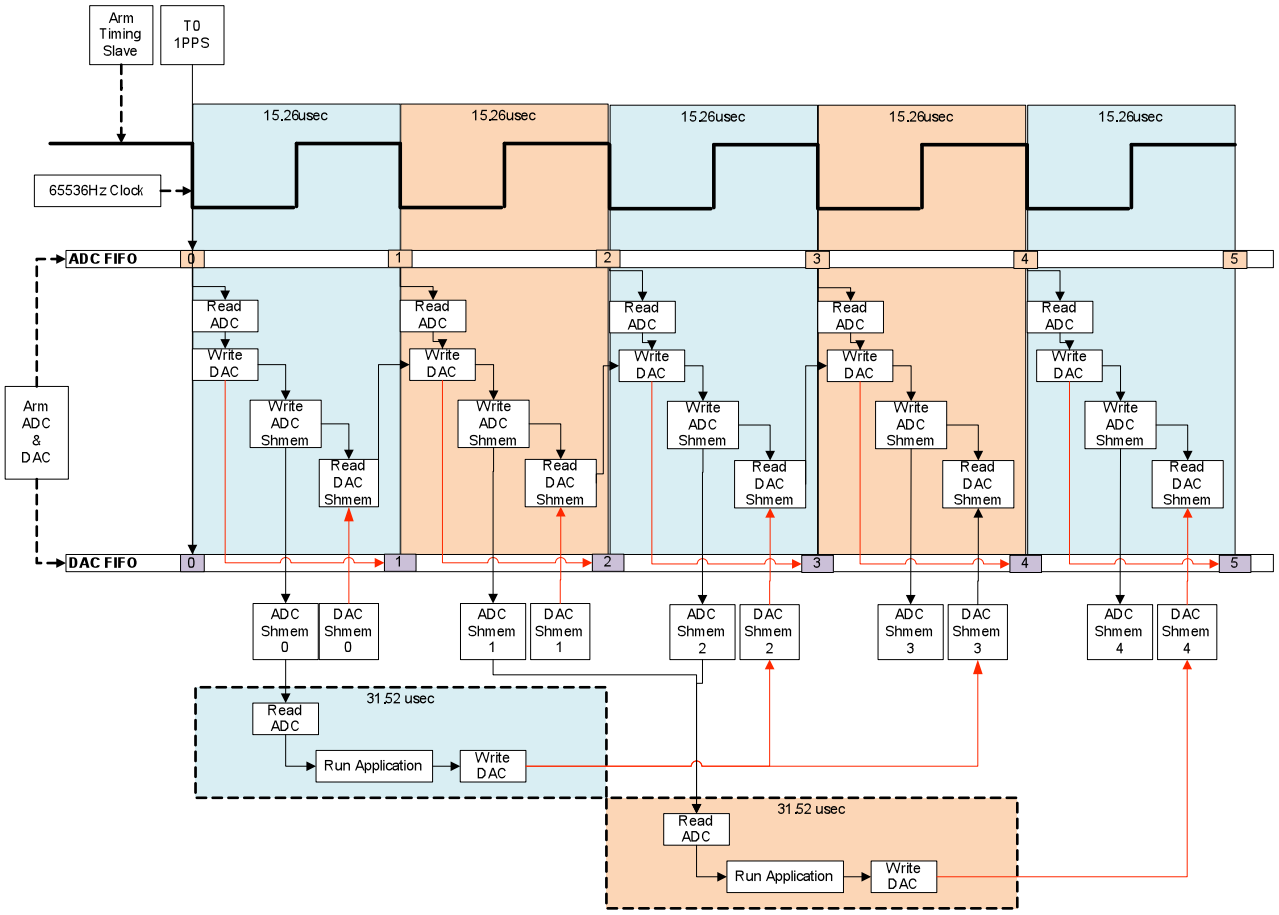


Figure 7: IOP and Slave Timing (Slave at 32768 Samples/sec)

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).

After initialization, the slave task waits for the first ADC memory block 0 to contain a GPS timestamp and a cycle count of "0", indicating that the IOP has just read data coincident with the GPS 1 Hz marker. The slave application now enters its infinite control loop. A couple of items of note:

1) At startup, only one ADC value is read from memory to trigger a code cycle (Cycle 0). This ensures, regardless of sample rate, each application starts its cycle 0 at the same point in time. Thereafter, the number of ADC samples to read is dependent on the defined code sample rate. In the example above, the 32K sample/sec slave will read two ADC values prior to triggering its next loop through the control code.
2) 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). How far ahead the slave application writes in the DAC output memory is a compile option. For example, by default, a slave process running at 2KHz would write 32 blocks ahead. However, if the slave process is verified to always run at less than the maximum of 488usec, this write ahead could be reduced, which in turn would reduce the phase delay in that control application.

8.3 IOP Processing Loop

The IOP processing loop is depicted in the following flow chart. This is similar to that described previously for a stand-alone application. Some differences:

1) 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.
2) 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.
3) 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.

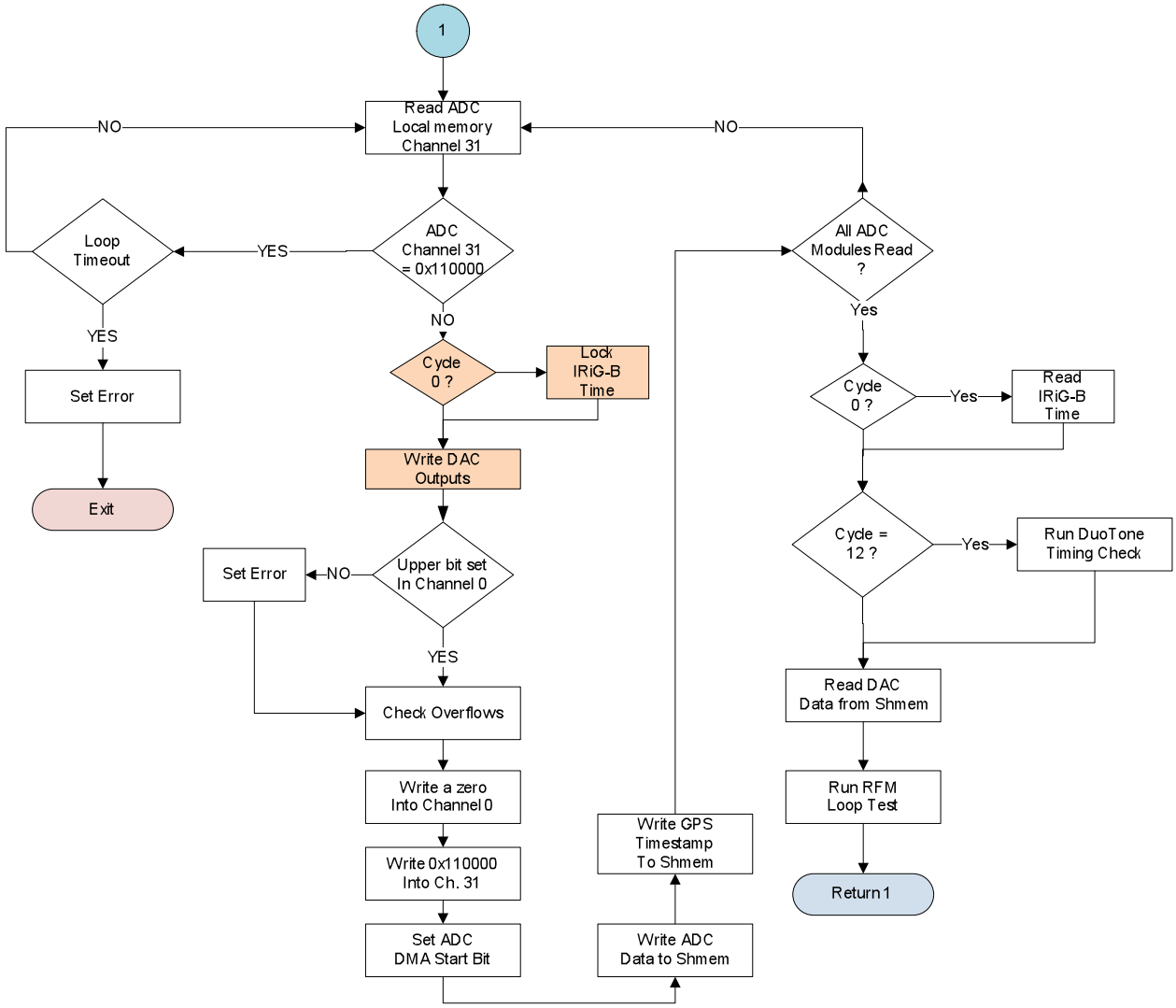


Figure 8: IOP Flow Chart

8.4 Slave Application Processing Loop

The slave application processes in a fashion similar to that described for a stand-alone application. Some distinct differences:

1) Does not directly map any I/O modules during initialization. Instead, it receives ADC/DAC shared memory pointers and base address locations for other I/O modules from the IOP.
2) ADC data checking (timing, first channel, etc.) is performed by the IOP. The slave applications uses the timestamp and cycle count in the ADC shared memory area to trigger its cycle.

9 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 usec), 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.