



*LIGO Laboratory / LIGO Scientific Collaboration*

LIGO-T0900607-v1

*LIGO*

12/1/09

---

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](mailto: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](mailto: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

<b>1</b>	<b><i>Introduction</i></b> .....	<b>3</b>
<b>2</b>	<b><i>References</i></b> .....	<b>3</b>
<b>3</b>	<b><i>Overview</i></b> .....	<b>3</b>
<b>4</b>	<b><i>RTS Software Requirements</i></b> .....	<b>4</b>
<b>5</b>	<b><i>RTS Design Overview</i></b> .....	<i>Error! Bookmark not defined.</i>
<b>6</b>	<b><i>Realtime Sequencer Overview</i></b> .....	<b>6</b>
<b>7</b>	<b><i>Sequencer Initialization</i></b> .....	<b>8</b>
<b>7.1</b>	<b><i>ADC Initialization</i></b> .....	<b>8</b>
<b>7.2</b>	<b><i>DAC Initialization</i></b> .....	<b>10</b>
<b>8</b>	<b><i>Sequencer Runtime</i></b> .....	<b>11</b>
<b>9</b>	<b><i>Timing Diagnostics</i></b> .....	<b>13</b>
<b>10</b>	<b><i>Timing Tests</i></b> .....	<i>Error! Bookmark not defined.</i>

## 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.

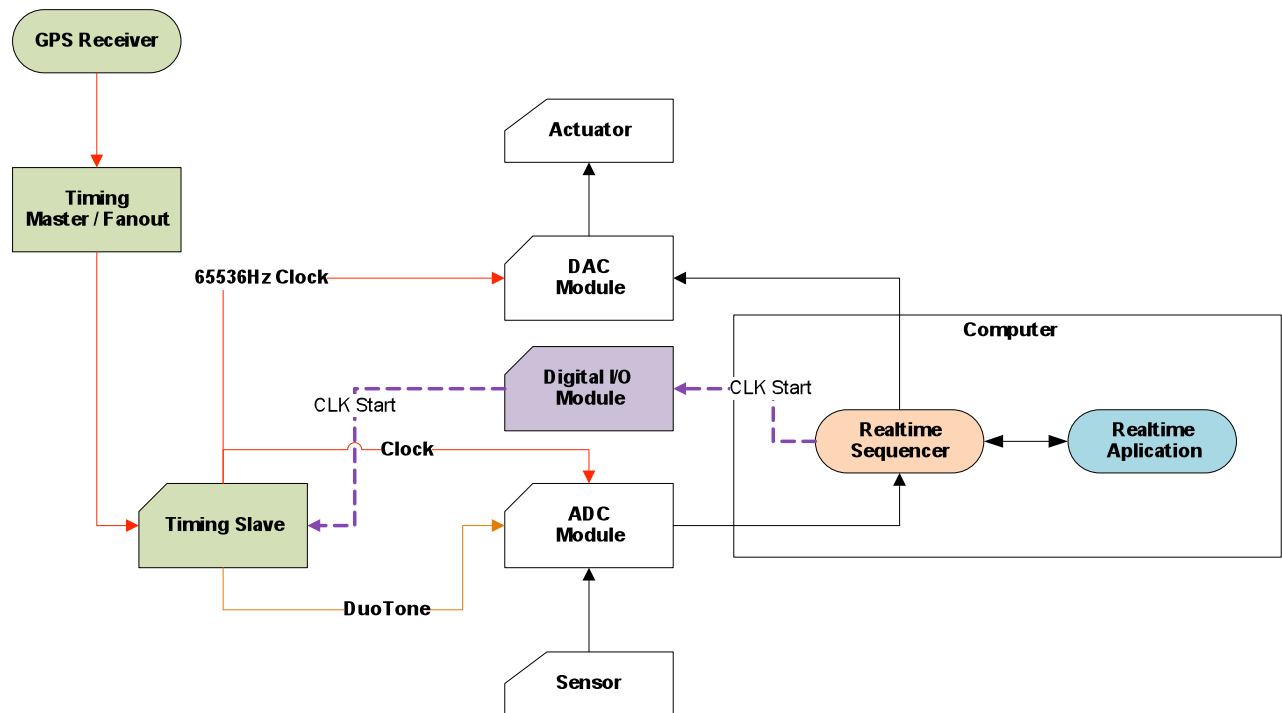
## 2 References

- 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 DOC TBD.

## 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:

- 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 ( $\geq 32768$  or  $\leq -32768$  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.

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.

The basic flow diagram for the sequencer and representative timing diagram are shown in the following figures. Further details are provided in following sections of this document. The source code for the sequencer is located in the *cds/advLigo/src/fe/controller.c* file.

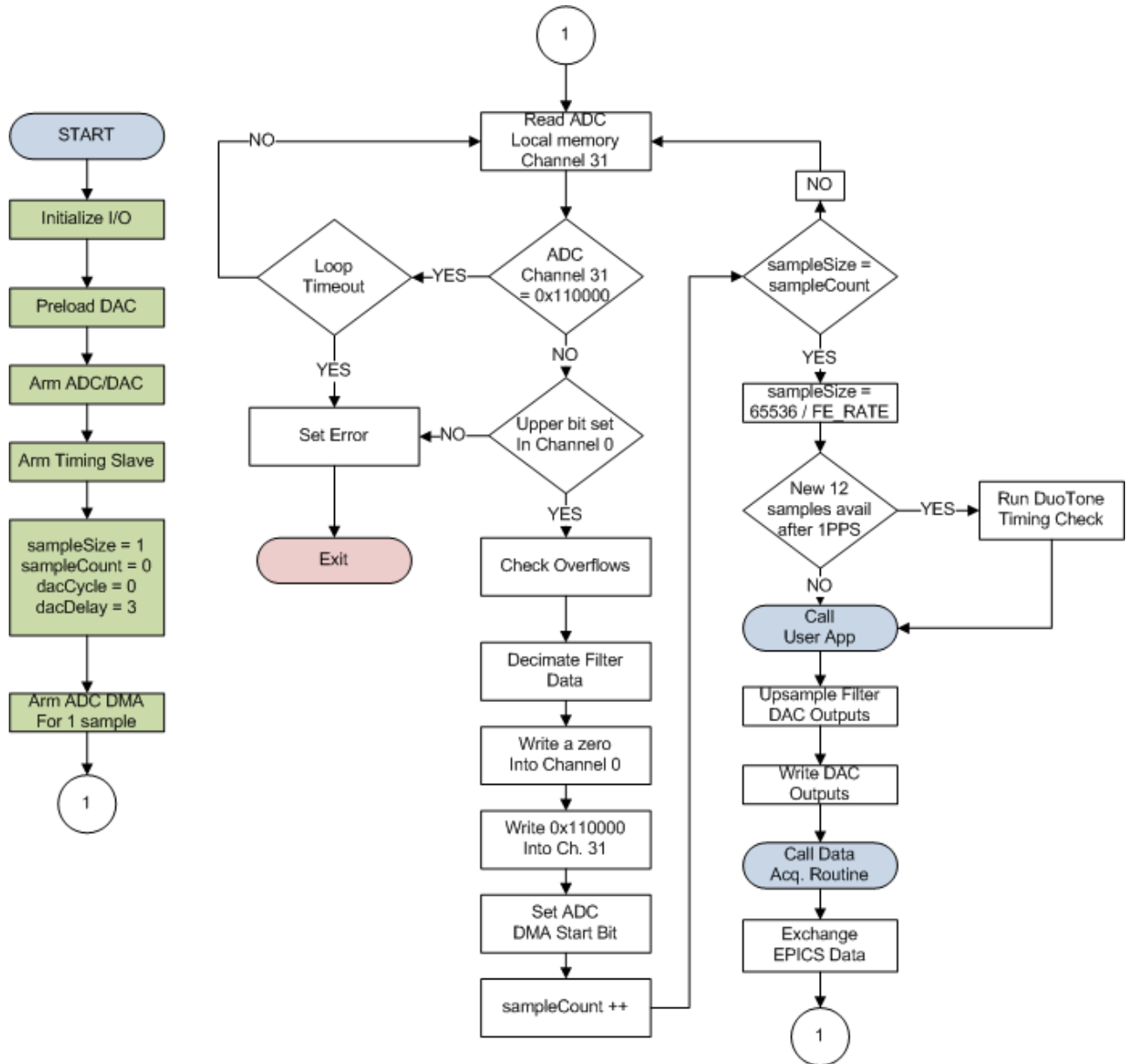


Figure 2: Sequencer Code Basic Flow Diagram

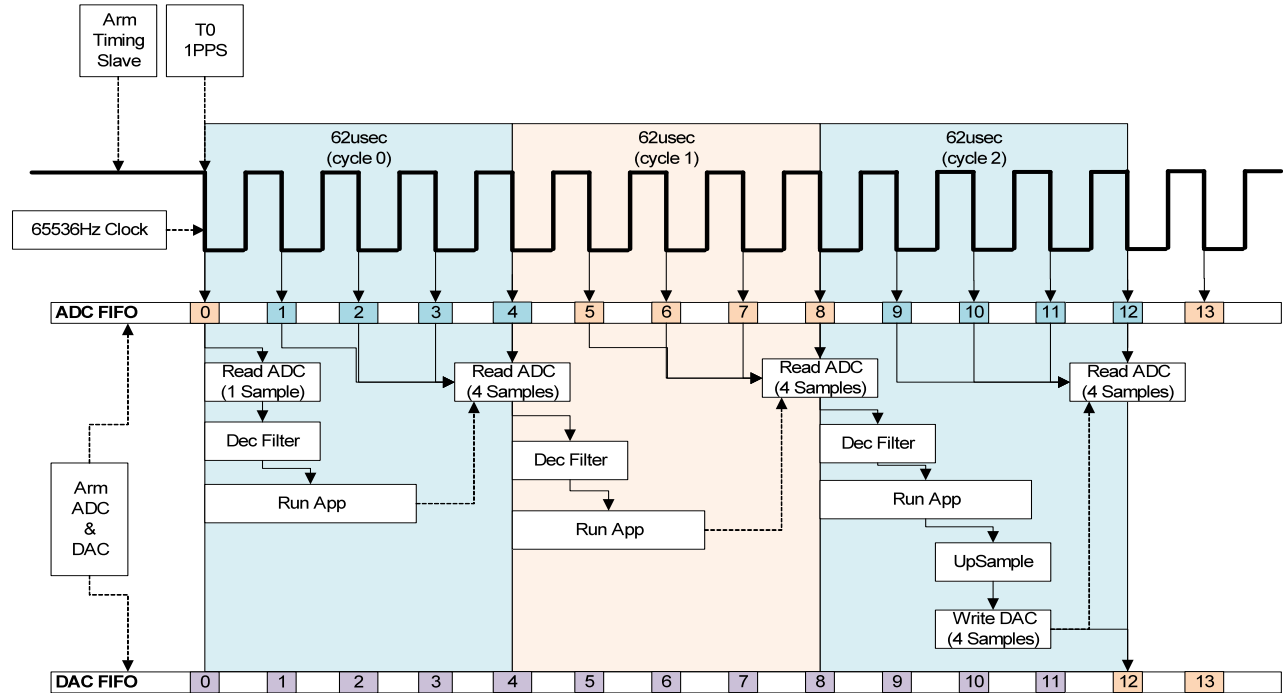


Figure 3: 16834Hz System Timing

## 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.

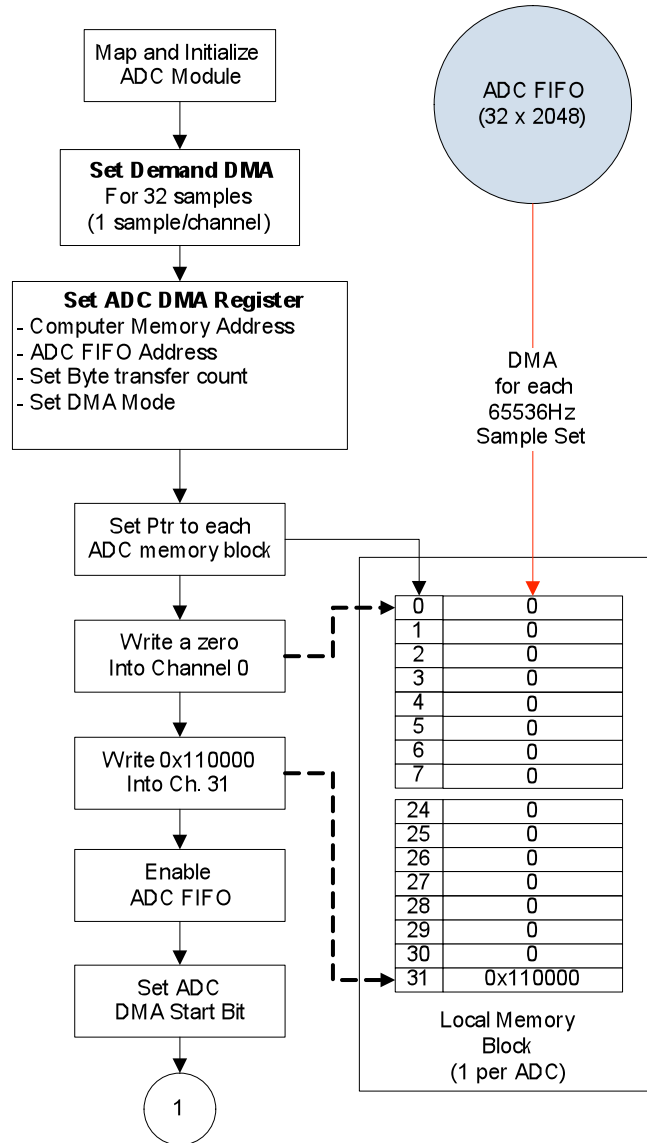
To enhance I/O performance, these modules have a capability known as “Demand DMA Mode”. In this mode, whenever an ADC FIFO contains  $\geq$  the user defined number of samples, and the “DMA Start Bit” has been set, the ADC will automatically transfer the defined number of samples to the user specified 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.

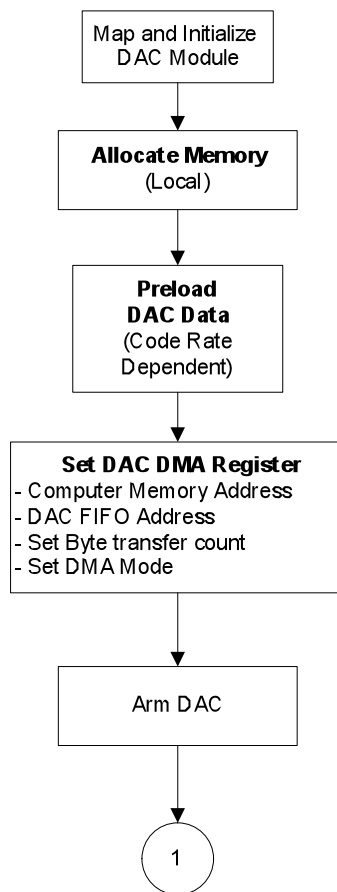


**Figure 4: ADC Initialization**

## 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  $65536/\text{FE\_RATE}$  (where  $\text{FE\_RATE}$  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  $(\text{FE\_RATE} - 1)$ . 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 or a slave. If it is compiled as a master, the RTS handles all defined 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 a master, a simple application eg filter module per ADC channel, could run at 16K samples/sec (or faster) and provide I/O for all other applications on that computer.

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.

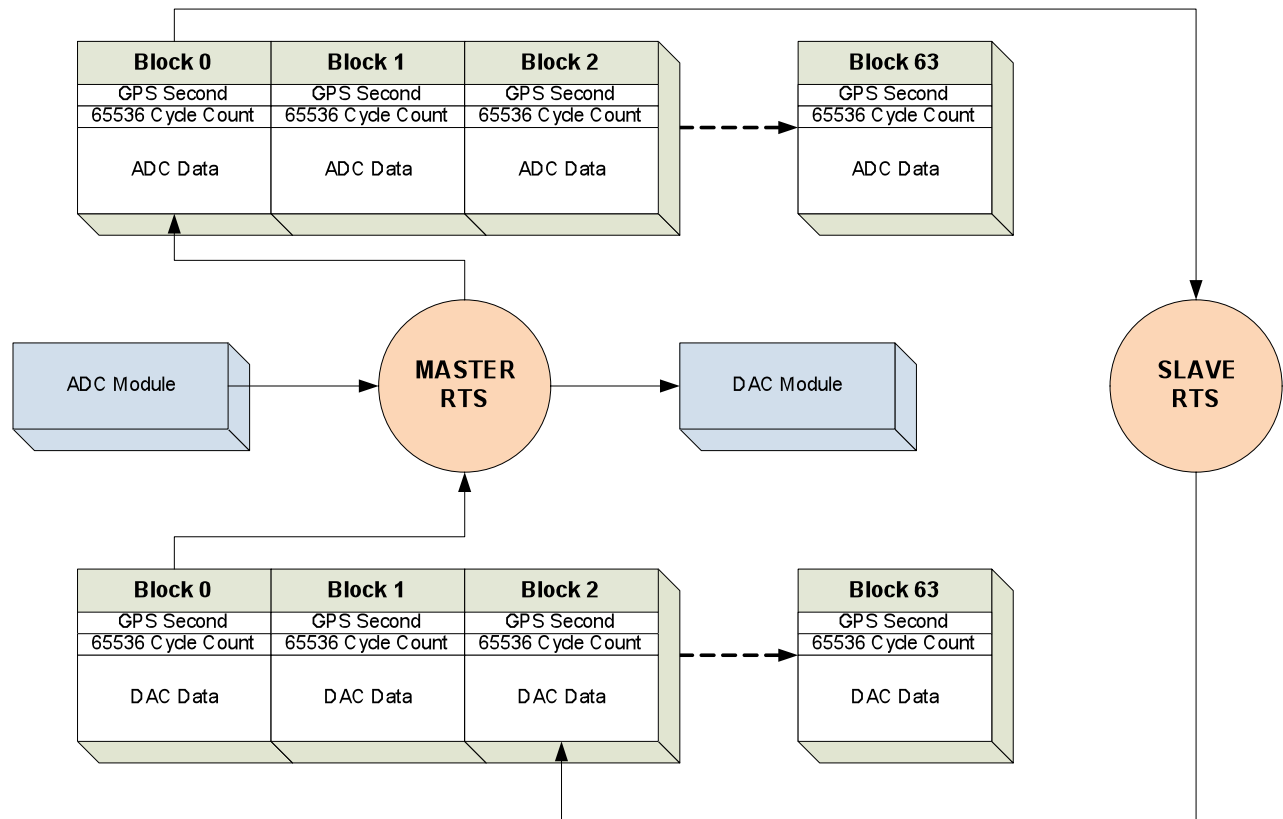


Figure 6: Master/Slave Operation

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

## 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 usec, from the 1PPS mark is passed on to an EPICS channel.

- 5) 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.
- 6) Longest time (in usec) for a single code cycle since last 'DIAG RESET' executed by an operator.
- 7) Time it takes to run the user application part of the code.
- 8) 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.
- 9) 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.