LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY

-LIGO-

CALIFORNIA INSTITUTE OF TECHNOLOGY

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LIGO CDS
Realtime System
Common C Code Libraries

R. Bork

Distribution of this draft:
This is an internal working note of the LIGO Laboratory

California Institute of Technology
LIGO Project – MS 18-33
Pasadena, CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project – MS 20B-145
Cambridge, MA 01239
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

www: http://www.ligo.caltech.edu/

Table of Contents

1 Purpose ... 4
2 Scope ... 4
3 References ... 4
4 Acronyms ... 4
5 Overview ... 5
5.1 CDS Architecture ... 5
5.2 Front End Software Sequencing ... 6
5.3 Front End Code Modules ... 7
5.3.1 Code Categories ... 7
5.3.2 Code Compilation ... 9
6 VME Hardware Initialization and Access Software ... 9
6.1 CDS Supported VME Modules ... 9
6.2 VME Hardware Initialization ... 9
6.2.1 Basic Startup Procedure ... 9
6.2.2 VME Initialization Specifics ... 10
6.3 VME Hardware and Front End Software Synchronization ... 12
6.4 Additional VME Drivers ... 12
6.4.1 GPS ... 12
6.4.2 ICS115 DAC Module ... 13
6.4.3 Xycom Decode ... 13
6.4.4 Linked List DMA ... 13
7 PCI Hardware Initialization and Access Software ... 14
7.1 RFM Routines ... 14
7.2 U232 Device Routines ... 14
8 Standard Filter Module (SFM) ... 15
8.1 Overview ... 15
8.1.1 SFM Code Module ... 15
8.1.2 SFM Filter Design and Communications ... 17
8.2 SFM Code Details ... 19
8.2.1 Data Structures ... 19
8.2.2 Required Headers and Source Files ... 21
8.2.3 SFM Usage in FE Code ... 22
8.3 Code Compilation ... 23
9 DAQ/GDS Software ... 24
9.1 Overview ... 24
9.1.1 GDS and DAQ Files ... 25
9.1.2 RFM Network Layout ... 26
9.2 Data Acquisition Software Library ... 27
9.2.1 Software Functional Description ... 27
9.2.2 Code Usage ... 29
9.3 Global Diagnostics ... 30
9.3.1 Overview ... 30
9.3.2 Functional Description ... 30
9.3.3 Code Usage ... 33
10 Building EPICS Systems ... 35
10.1 Automatic code and database generation ... 35

10.1.1 Perl Script ... 35
10.1.2 Input Files ... 35
10.2 Epics Sequencers (.st files) ... 37
10.3 Epics Database (.db) Files ... 37
10.4 Epics Build ... 37
11 Software Installation ... 38

1 Purpose

The purpose of this document is to describe the computer software libraries common to all LIGO Control and Data System (CDS) realtime front end systems. CDS realtime front end systems are defined as those CDS VME based processor systems which must run synchronously at either 16384Hz or 2048Hz to perform LIGO control functions and/or LIGO data acquisition functions.

2 Scope

The scope of this document includes the following:

- An overview of the CDS hardware architecture as it pertains to realtime controls.
- A description of standard CDS software libraries employed in realtime controls. This includes functional descriptions, code usage and code compilation.
- Software to interface in realtime between front end controllers and the LIGO Global Diagnostic System (GDS) and Data Acquisition System (DAQ).
- A description of the non-realtime, asynchronous software interfaces to the realtime controls.
- Basic CDS software installation guidelines and checklist.

Specifically excluded from the scope of this document:

- Specifics of individual realtime front end code.
- Detailed CDS hardware descriptions and drawings.
- CDS software which does not run on realtime systems, such as those that are typically referred to as auxiliary systems.

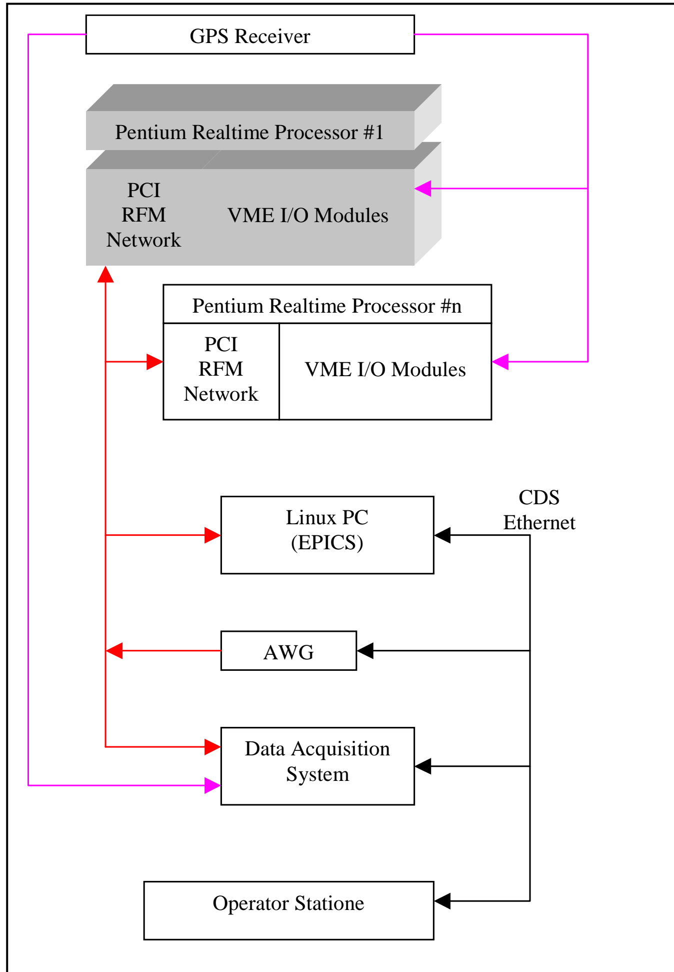
3 References

4 Acronyms

5 Overview

5.1 CDS Architecture

All LIGO CDS realtime front end systems are built in VME with Pentium processors and various Input/Output (I/O) modules. All of these systems must run synchronous with each other and operate at either 16384Hz or 2048Hz.



A simplified overview of the LIGO CDS architecture is shown at left, with a realtime front end crate highlighted.

In addition to a VME backplane connection to various I/O modules, the front end processors also have one or more PCI bus connections. These are used to install Reflected Memory (RFM) realtime networking modules. The RFM networks are used to interconnect and move control signals between multiple front end processors. These RFM networks also provide front ends with connections to the Data Acquisition (DAQ) system, Global Diagnostics System (GDS) Arbitrary Waveform Generator (AWG) and Linux PCs.

GPS receivers provide for synchronous operation of the CDS components. This is done by sending 1Hz and $2^{\wedge 22}$ Hz clock signals to time the sampling of VME ADC and DAC modules. The front end software is then slaved to the VME hardware sampling rate.

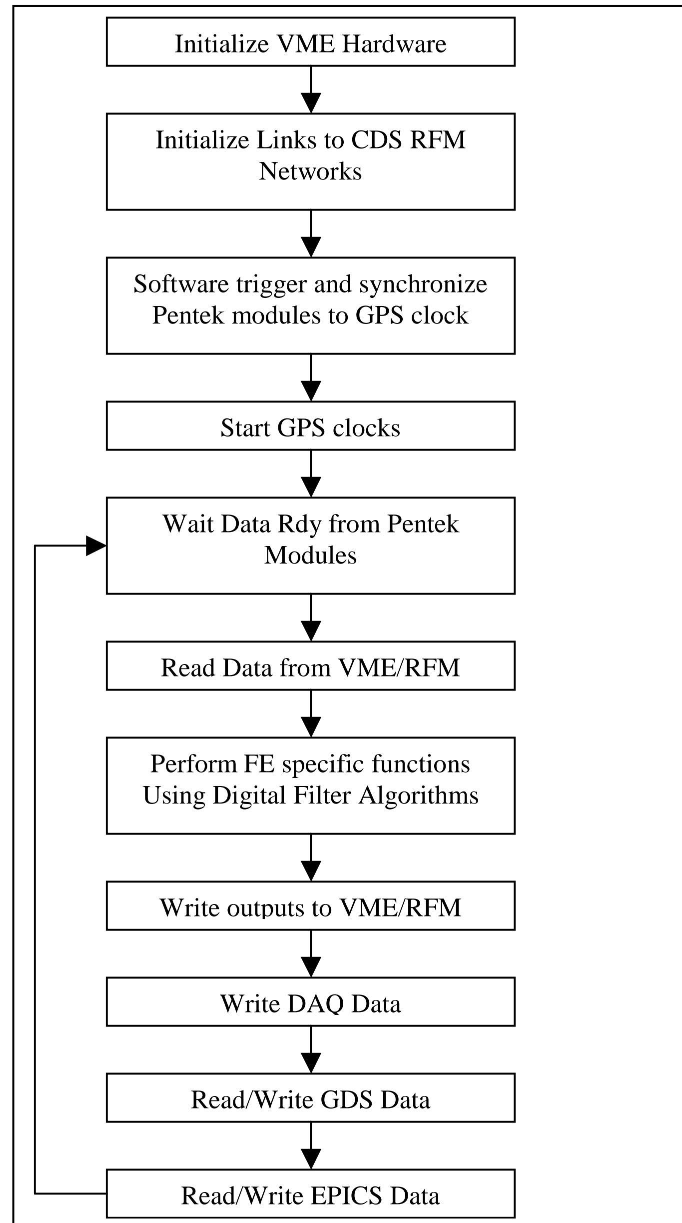
The DAQ system provides for the synchronous collection and storing of data. Data is acquired at rates up to 16384Hz and then stored to disk in a LIGO standard format, known as data frames. This data is then available via the CDS Ethernet to operator stations and directly from disk to the LIGO Data Analysis System (LDAS).

The GDS AWG provides the capability to inject test signals into CDS front end processors. This is done to run various diagnostics, including coherence and transfer function calculations.

The Linux PCs run EPICS and provide the interface to CDS operator stations via a connection to the CDS Ethernet. Common software and development tools for EPICS is described in the last section of this document.

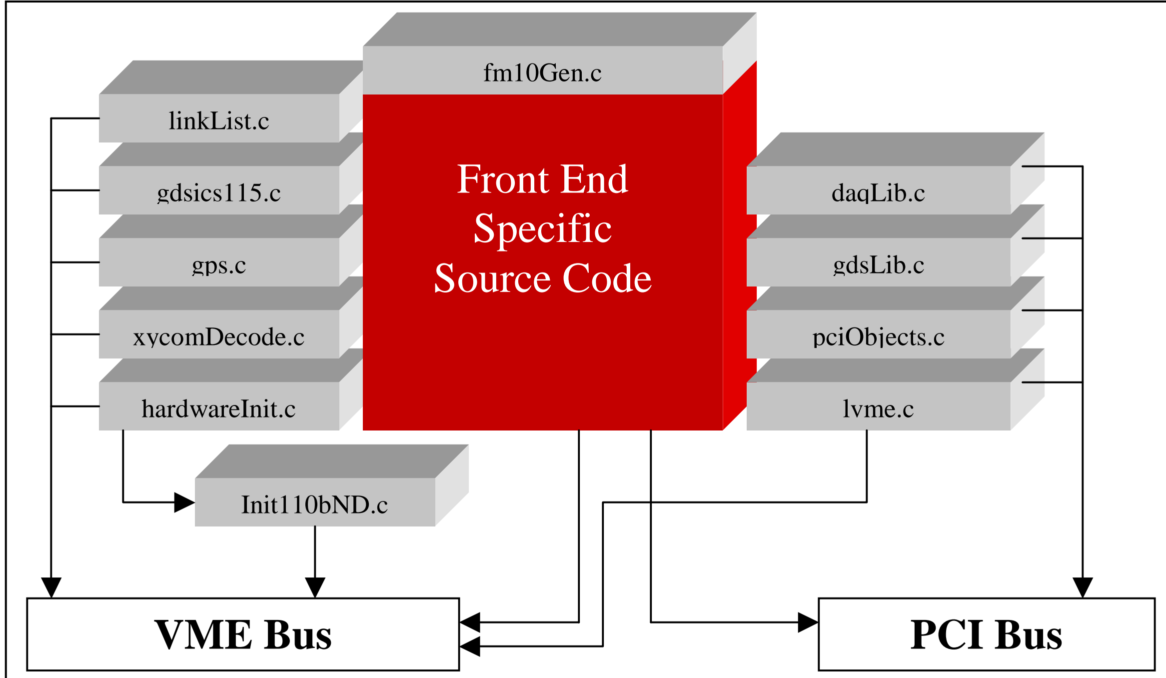
5.2 Front End Software Sequencing

The software within the VME realtime front end processors has evolved into a fairly standard sequence, as shown in the basic flow chart below. After initializing hardware, including the VME modules and Reflected Memory (RFM) realtime network connections, the front end code modules go into an infinite loop. This loop is synchronized by the GPS clocks being applied to the Pentek ADC/DAC modules. Whenever these modules indicate data ready, at either 16384Hz or 2048Hz, the control loop is entered and run. When the control loop is complete, the front end again waits for a data ready signal from the VME ADC module.



5.3 Front End Code Modules

To facilitate code development and testing of these front ends, a number of C code routines have been developed over time to handle common front end tasks. The front end source code files which contain these routines are depicted in the following figure. All of these source code files may be found in the LIGO CVS tree at *cds/rts/src/include/drv*.



5.3.1 Code Categories

For purposes of this document, the code libraries are discussed in following sections as being in one of four categories:

- Operating System
- VME Hardware Initialization and Access
- PCI Hardware Initialization
- Data Acquisition
- Standard Control Algorithms

5.3.1.1 Operating Systems

Up until the past year, all FE processors used vxWorks as the operating system software. This is still true of all FE systems which employ realtime interrupts to trigger processing. However, many systems have now been switched over to Linux, a trend which will continue in the future. When Linux is used, the FE specific software is loaded into the Linux kernel and essentially turns off Linux and takes control of the system. Therefore, Linux is primarily just used to boot the processor, load the software and provide some memory mapping routines for the FE software. Whereas vxWorks provided for direct memory mapping and absolute addressing, Linux provides for virtual memory mapping. This required some code to provide this mapping to the existing FE software. This lead to the routines included in lvme.c, further described below.

All FE processors are booted via the CDS Ethernet. In the case of those running vxWorks, a boot table is maintained in NVRAM. On power up or reset, the processor connects to the CDS server to obtain the vxWorks runtime kernel.

Linux processors use PXE to boot up. A Linux DCHP/TFTPD server resides on the CDS network to provide the network address and Linux kernel for the FE processor.

Once booted, the vxWorks processors provide for a telnet connection to load and start FE specific software. The Linux processors provide a secure shell (ssh) connection for this purpose. For both operating systems, startup files may also be provided to automatically load and start code on reboot. Once the FE specific software has started, all Ethernet connections are turned off by the FE software, with communications only allowed via the RFM networks.

5.3.1.2 VME Hardware Initialization and Access Routines

This category of software includes:

- hardwareInit.c : A set of routines for the initialization and startup of CDS standard VME modules. Later discussion of this software will include init110bND.c, which is a subset of hardware initialization software specific to ICS110B ADC modules.
- linkList.c : A set of routines for performing linked list DMA between CDS realtime Pentium processors and CDS supported VME modules.
- gdsics115.c : Code specifically for the operation of ICS115 DAC modules.
- gps.c : Code routines for use with CDS Global Positioning System (GPS) VME receiver modules.
- XycomDecode.c : Works in conjunction with CDS standard filter modules to define and synchronize the switching of hardware and software filters.
- lvme.c: This software was initially developed to provide kernel insertion, memory mapping and VME I/O capabilities in Linux. However, this software may now be used for both Linux and vxWorks systems. This will gradually replace older code, such as hardwareInit.c and linkList.c, in all systems as it provides the capabilities of that software and more.

5.3.1.3 PCI Hardware Initialization Routines

CDS employs reflected memory modules (RFM) on the PCI bus of VME front end processors for realtime communications between various front ends, to send data to the Data Acquisition (DAQ) system, to receive data from the Global Diagnostics System (GDS) Arbitrary Waveform Generator (AWG) and communicate to operator stations via Linux PCs. The interface to VME is also through a Universe 232 chip located on the PCI bus. These PCI devices are automatically memory mapped by the CPU at boot. In order for the front end to gain access to these devices, a set of routines have been developed to return these addresses and perform a few other functions through the PCI bus. These routines are found in the *pciObjects.c* file for vxWorks systems and *lvme.c* for Linux systems.

5.3.1.4 Data Acquisition Software

All CDS front ends must provide data acquisition and remote testing capabilities. These functions are provided in the following two source code files:

- daqLib.c : Performs all of the necessary data acquisition tasks.
- gdsLib.c : Performs the diagnostic tasks of writing front end test points to the DAQ system and reading excitation channels from the AWG.

5.3.1.5 Standard Control Algorithm Software

All CDS front end systems which perform control functions do so with user defined Infinite Impulse Response (IIR) digital filters, which are arranged to provide the proper transfer function for a specified number of control degrees of freedom. The implementation of these filters in done through a standard set of software provided in the *fm10Gen.c* file.

5.3.2 Code Compilation

The specifics of compiling these code files is covered later in this document. However, it should be noted here that none of these files are presently written to be compiled stand alone and runtime linked into front end processors. In fact, there is no Makefile in the *cds/rts/src/include/drv* area. Instead, each desired C code file must be included in the specific front end source code and compiled with that code. While this is somewhat unusual in C code development, this was done to allow for both modular maintenance of software and providing "inlining" of code for compiler optimization and resulting code performance enhancement.

6 VME Hardware Initialization and Access Software

All CDS realtime systems operate on a Pentium III or Pentium 4 processor in a VME crate, employing various VME I/O modules. CDS has standardized on a set of VME boards, which are supported in various software routines.

6.1 CDS Supported VME Modules

CDS software supports the following standard VME modules:

- Pentek 6102 ADC/DAC: Eight channels each of individual, simultaneous sampling 16 bit ADC/DAC.
- ICS110B ADC: 32 , 24bit sigma-delta ADC channels.
- ICS130 ADC: 32, 16bit sigma-delta, 1.2MHz
- Xycom 220: 32 channels of binary output.
- Xycom 212: 32 channels of binary input.
- Frequency Devices DAC (FD8DAC): 8 channels of 24 bit DAC outputs.
- ICS115 DAC: up to 32 channels of 24 bit sigma-delta DAC outputs.
- Clock Driver Module (CDM): LIGO developed timing module based on 1Hz and $2^{\wedge 22}$Hz clocks derived from GPS receivers.
- Brandywine GPS receivers and IRIG-B slaves: CDS uses both a standard GPS receiver and one specifically designed and produced for LIGO which outputs 1Hz and $2^{\wedge 22}$Hz clocks.

For the most part, the code specific to a particular front end handles I/O to/from VME directly ie does not call a standard routine or device driver. This is primarily for speed considerations. However, all VME hardware requires some initialization on reboot and a means to begin operation synchronously with the GPS clocks. This is done through a standard set of call described in the following subsections.

6.2 VME Hardware Initialization

All of the routines required by CDS realtime control front ends to initialize their VME hardware and startup synchronous to the CDS GPS timing system are included in cds/rts/src/include/hardwareInit.c (for vxWorks) or *cds/rts/src/include/hardwareInitLinux.c* (for Linux and vxWorks). All front end code will eventually switch over to the latter, therefore this is the code that will be described here. This source file is automatically included from *lvme.c*.

6.2.1 Basic Startup Procedure

All realtime front ends must initialize their VME modules and synchronize themselves to the CDS GPS clocks in the same fashion. This involves the following sequence of calls:

- mapVme(): If a FE is running on Linux, this call is made automatically when the software is installed into the kernel. FE code on vxWorks requires this call be made specifically on startup. This software will provide memory mapping of all VME modules found in the crate and provide a set of global pointers.
- hardwareInit(): Verifies the existence of and performs initialization of all VME modules to be used in a front end crate.

- vmeLinkList(): If a front end uses linked list DMA to read/write data from/to its VME modules, it must set up the DMA linked list. This may either be done through specific front end code or by making a call to this initialization routine.
- startPenteks(): This routine will set interrupt masks (if used) and software trigger the Pentek modules.
- syncPenteks(): Routine to synchronize the Pentek clock downcounters to the CDS GPS system.
- adcClock(ENABLE_CLOCK): Start the CDM, which will in turn start producing the $2^{\wedge 22}$Hz clocks on the next 1Hz GPS signal. This synchronizes the start of the front end with all other CDS systems.

6.2.2 VME Initialization Specifics

6.2.2.1 mapVme()

This software provides for two primary functions:

- If used in a Linux system, it opens and maps memory windows to the various supported VME modules.
- It sets up and provides a set of global pointers to the base address of the supported VME modules.

For memory mapping, the code opens windows into A16, A24 and A32 space. This space is limited in that A24 is set up primarily for ICS110B modules, starting at 0xa00000, and A32 space is defined to start at 0x60000000 in support of Freq. Dev. DAC modules.

After opening the memory windows, if necessary, the code will initialize pointers to all of the VME modules found in the crate. These are global variable pointers of type defined for each type of VME module supported. A global structure (vmeList) will also be updated with the number of each type of VME module found in the crate by this software.

6.2.2.2 HardwareInit()

The hardwareInit() routine will perform initialization of all VME modules within a crate. The routine will return a –1 if a fault is encountered in the initialization, or zero if initialization runs properly.

The call to hardwareInit() requires the passing of three parameters:

- VME_ASSIGN *vmeMod: This structure provides a place for the calling program to list the number of each type of supported VME modules that the code expects to run.
- Int masterMode: If the system is to use ICS110B ADC modules and the first module is to run as a Master (see ICS110b manual), then this variable must be set to one. If the first module is to be run as a mid or end slave, then this variable must be set to zero.
- ICS110B_ADC_GAIN *gains: ICS110B1 modules have the variable gain setting capabilities. If these gains are to be set, then this structure must be filled with the desired gain settings. If a NULL pointer is passed, then all gains are set to one by the initialization routine.

Once called, this routine runs through the following sequence:

1. Initializes ICS110B modules. This includes ADC calibration, setting acquire and clock sources, master/slave modes, and loading channel gain settings.
2. Initializes Freq. Dev. DAC modules. This includes setting the clock source and initializing all outputs to zero.
3. Initializes all Pentek 6102 modules. This includes setting up the internal clock down counters for either 2048Hz or 16384Hz operation.

It should be noted here that the specific front end application is responsible for all VME module accesses after module initialization, with the possible exception of linked list DMA. In the latter case, library routines are available and may be used (see section 5.4). Since all VME (and PCI) modules are memory

mapped to the CPU, no traditional style I/O drivers are provided. This was specifically decided upon to improve system performance. However, to facilitate VME access, the hardware initialization routines set up some global pointers to I/O modules for the specific front end code to use. These pointers are:

- struct P6102 *pmodule[]: This pointer describes the various registers of the Pentek 6102 ADC/DAC modules, as defined in *cds/rts/include/drv/pentek6102.h*.
- struct XYCOM212 *xycom212[]: Pointers describe the register map of the Xycom 212 binary input module, as defined in *cds/rts/include/drv/xycom.h.*
- struct XYCOM220 *xycom[]:Pointers describe the register map of the Xycom 220 binary output module, as defined in *cds/rts/include/drv/xycom.h.*
- fdDAC *pDAC[]: Pointer to Frequency Devices DAC module registers, as defined in *cds/rts/src/include/drv/vm8dac.h.*
- struct ics110b *pIcs110b1[]: Pointer to ICS110B module registers, as defined in *cds/rts/src/include/drv/ics110b.h.*

6.2.2.3 vmeLinkList()

Many of the CDS FE applications now make use of the linked list DMA capabilities of the Universe 232 chip on the VME processor boards. What this allows is for VME transfers to occur without CPU intervention or control at the same time that the front end CPU performs calculations. This results in increased performance. In fact, some front ends could not complete their calculations in the prescribed time without this capability.

The basic concept is that the U232 chip DMA register settings are set up in CPU memory space, with a pointer from one set of settings to another. Once the command is sent to the U232 chip to start DMA, it will read in these settings, one at a time, perform the described DMA, set a flag indicating that a particular DMA has completed, and then load the next set of DMA settings. This is done until an end of list flag is found.

What the vmeLinkList() routine does is set up this list of DMA commands in local memory and properly initialize the U232 chip DMA and control registers to make use of this list. This list is automatically generated when this routine is called with the initialization flag set. The list is developed such that all Pentek ADC modules are read first, in address order, followed by all ICS110B ADC modules. Once the list is developed, the list starting point information is sent to the U232 VME interface chip, this interface is initialized, and DMA is ready to operate on command. Initiating the DMA transfer and other functions of this routine are described in 5.4.

6.2.2.4 startPenteks()

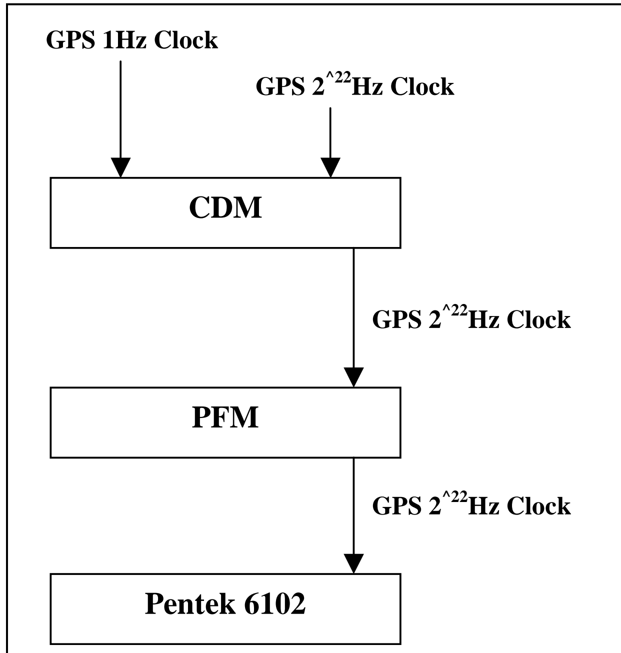
This routine will set up the interrupt vector and enable interrupts from the first Pentek module (if code is defined with INTR_DRV set). It will also reset the ADC FIFOs and software trigger the Pentek modules. Note that this code should be called with the CDM turned OFF (done by the hardwareInit() routine).

6.2.2.5 syncPenteks()

Prior to startup, the Pentek module down counters must be synchronized to the GPS 1PPS clock such that the ADC sample occurs synchronous to this clock. This routine provides that function.

6.3 VME Hardware and Front End Software Synchronization

All CDS VME realtime front end systems must run synchronously with the CDS GPS timing system. This is done by driving the Pentek 6102 modules with the GPS clocks and having the front end software wait until the first Pentek module within a VME crate indicates that data is ready. The hardware scheme is shown in the figure.



The GPS 1Hz and 2^{22}Hz clocks are cabled to the front panel of the CDM via a four pin LEMO connector. After all VME hardware has been initialized, the CDM is set to OFF by the front end software. The previously described routine startPenteks() is called, which software triggers the Pentek modules to start sampling. However, since the CDM is turned OFF, no clocks are now being supplied to the Pentek modules and they will not sample at this point.

The front end code then turns ON the CDM. This will cause the CDM to begin relaying 2^{22}Hz clocks coincident with the next 1Hz GPS signal. This 2^{22} clock is fed to the Pentek Fanout Module (PFM) and then to the ADC clock input of all Pentek modules. The purpose of the PFM is simply to change the clock signal from ECL to TTL, required by the Pentek modules.

On the Pentek modules, the clock goes through the internal downcounters, previously initialized by the hardwareInit() routine, causing the modules to sample synchronously at 16384Hz or 2048Hz. All CDS front end software waits for a sample to be ready from the first Pentek within a crate. In this fashion, the software is slaved to the hardware and therefore synchronized at one of the two CDS sampling rates.

One problem encountered with this scheme is that the initial count within the Pentek clock down counters is unknown ie there is not a VME register available to check the down counter values nor a means to reset the down counters. Therefore, for a 16384Hz system, the first Pentek sample could occur anywhere from 0 to 61usec after application of the GPS clock.

To bring this uncertainty within the 10usec timing requirement, the syncPenteks() routine was developed and must be called just prior to enabling the CDM on code startup. What this code does is run the Pentek modules and attempt to determine when the down counters are almost at zero and then turn OFF the CDM. Then, when the main code again starts the CDM to start the main code thread, the Pentek modules will begin sampling within ~5usec of the GPS 1Hz clock pulse.

6.4 Additional VME Drivers

6.4.1 GPS

For timing, the CDS uses a number of custom GPS receivers. A standard library of calls have been developed for addressing these modules. These routines are primarily used by the DAQ controller software. These routines can be found in the *drv/gps.c* file.

6.4.2 ICS115 DAC Module

ICS 115 DAC modules are used for outputs in the AWG and various DAQ ADCU units. The file *drv/gdsics115.c* contains a number of initialization and runtime routines for these modules.

6.4.3 Xycom Decode

Typically, all LIGO control front ends must switch some analog hardware filters synchronously with digital filters. This is done by switching a bit in a Xycom digital output module. In some systems this is hard coded. In others, such as the LSC, this switching can be assigned dynamically via an EPICS interface screen. The front end code uses the *decodeXycom()* routines in the *drv/xycomDecode.c* file for this dynamic assignment and synchronous switching.

6.4.4 Linked List DMA

Some systems still perform VME linked list DMA using the routines found in drv/linkList.c. However, any new code development should use the routines provided in drv/lvme.c.

7 PCI Hardware Initialization and Access Software

The CDS VME processors contain up to three PCI Mezzanine Card (PMC) slots. These slots are populated with RFM network modules. The Universe 232 VME interface chip is also located and accessed on the PCI bus. Since the addresses of PCI modules are dynamically set up by the CPU on boot, a number of routines have been developed to locate these devices and return device information. These routines are located in *cds/rts/src/drv/pciObjects.c.*

7.1 RFM Routines

Two models of RFM are supported by CDS software. These are the VMIC5579 (400Mbit/sec) and VMIC5565 (2Gbit/sec). These networks contain 64Mbyte of memory. The lower 32 Mbyte is used for communications between front end processors and between front end processors and the Linux PCs running EPICS. The structure of the lower 32Mbytes is described in *cds/rts/include/iscNetDsc.h*. The upper 32Mbyte is reserved for transmission of data between the front ends, the DAQ system and the GDS AWG. The structure for the upper 32Mbyte is defined in *cds/rts/src/include/daqmap.h*.

Routines are provided to get information about these modules, including PCI register and base memory location addresses. These routines are:

- int is5565RfmCard(int instance): The *instance* variable is 0, 1 or 2, indicating which of up to three PMC slots to check. If a VMIC5565 module resides in that slot, this routine returns 0. If there is not a VMIC5565 module in this slot, it returns –1.
- UINT32 findRfmCard(UINT16 instance): The *instance* variable is 0, 1 or 2, indicating which RFM module to find. If that RFM module is found, this routine will return the memory base address of that module. If no module is found, the routine will return –1. This routine will search for both VMIC5579 and VMIC5565 modules.
- UINT32 findRfmCard5579(UINT16 instance): This call is similar to the previous routine, however it will only check for VMIC5579 modules.
- UINT32 findRfmCard5565(UINT16 instance, UINT16 locRegister): Routine specifically finds VMIC5565 modules. Depending on *locRegister* setting, code will return the memory base address or the DMA register base address. Code returns –1 if module not found.
- UIN32 rfmCardNodeId(UINT16 instance): Returns the value of the node ID register for the given RFM module.
- UINT32 findRfmCardPciAdd(UINT16 instance): Returns the PCI control register address of the selected RFM module.
- UINT32 findRfmData(UINT16 instance, UINT16 info): Returns the information on a RFM module based on the *info* parameter. This info requested may be the PCI busId, PCI device ID, or PCI address.

7.2 U232 Device Routines

The control registers of the Universe 232 VME interface chip are located on the PCI bus. The following routines provide address information:

- UINT32 findU2332Chip(UINT16 instance): Returns the base address of the U232 chip on the PCI bus.
- void vmeBusReset(): Writes a VME bus reset bit in the U232 control register. This routine is commonly called by front end systems to perform a reset on command from operators via the EPICS PC.
- void initSlave(UINT32 ptr): Configures a Pentium VME CPU to open its memory into the designated VME Slave memory space. This is typically used when multiple processors are collocated within the same VME backplane.
-

8 Standard Filter Module (SFM)

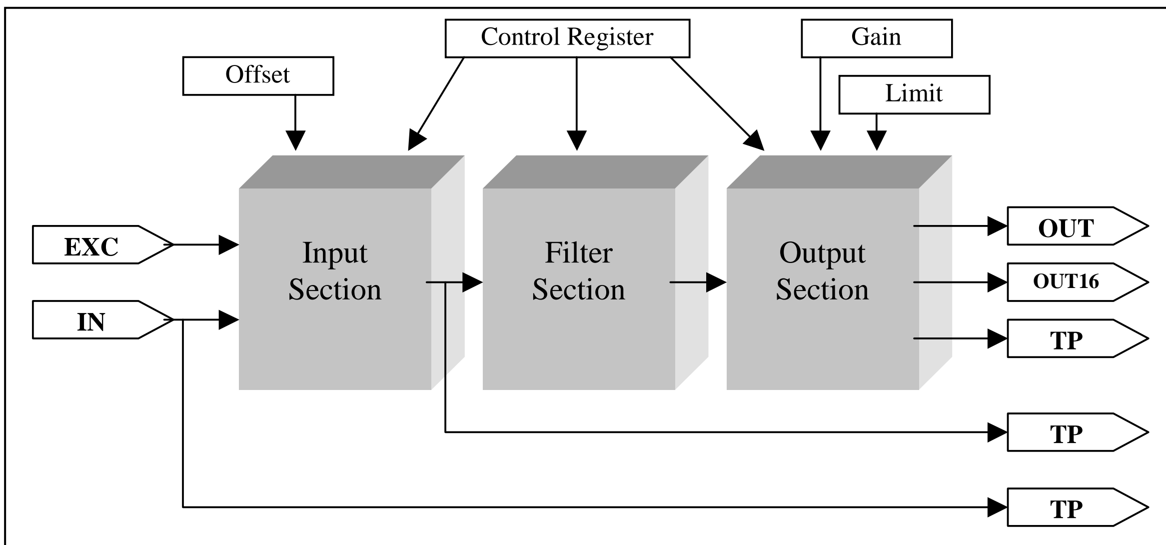
All CDS FE processors use digital Infinite Impulse Response (IIR) filters to perform a majority of their signal conditioning and control algorithm tasks. In order to facilitate their incorporation into FE software and to provide a standard set of DAQ and diagnostic capabilities, the Standard Filter Module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

8.1 Overview

8.1.1 SFM Code Module

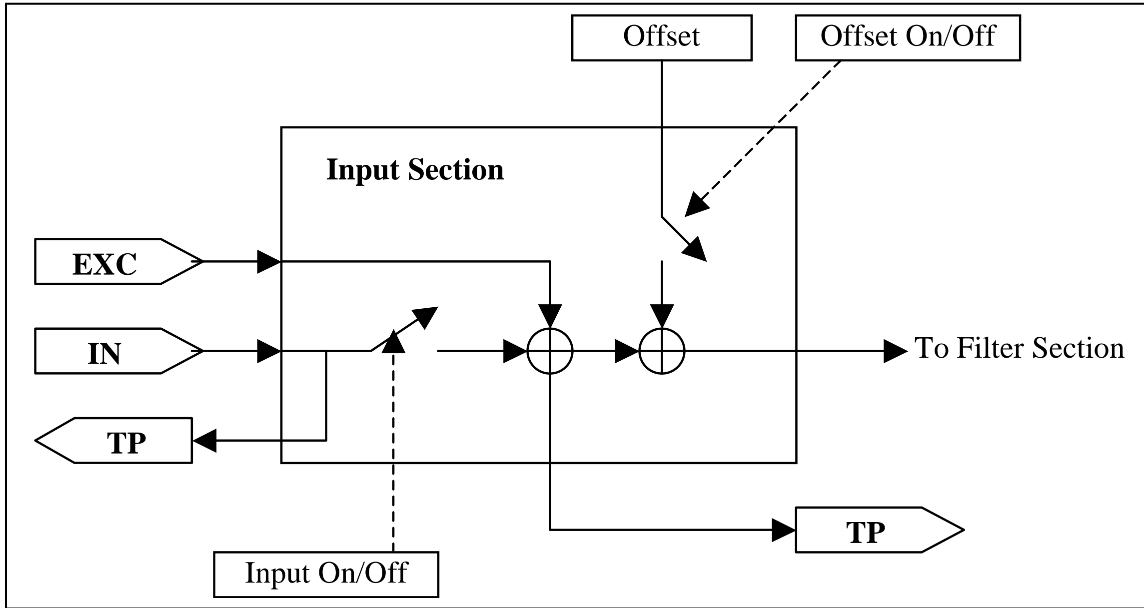
For purposes of description, the SFM is divided into three parts, are shown in the following figure. These sections are:

- Input Section: An SFM takes as input both a signal from the calling FE software and a selectable GDS EXC signal. The output of this section is the FE signal added to the GDS EXE signal, plus an operator input offset. Both the input from the FE code and the combined FE/EXC signal are available as GDS TP.
- Filter Section: Up to 10 digital filters may be defined for a single SFM. These are IIR filters with up to 10 Second Order Sections (SOS) each. An offset may be applied to the input of the filter section and a gain applied at the output. Each IIR filter may be individually turned on/off.
- Output Section: The output of the SFM is available to the FE software via the OUT channel. A gain and limit may be applied to this signal and the output may be turned on/off by the FE code. Along with the normal output, a filter decimated output (OUT16) is produced and a GDS TP is available. While the OUT and OUT16 channels are turned on/off by the FE code, the GDS TP is always on.



8.1.1.1 Input Section

A more detailed view of the SFM input section is shown in the following figure. It takes as inputs a signal from the calling FE code and a GDS Excitation (EXE) signal. Additionally, the input may be turned on/off and an offset applied under FE code control. Two TP are made available to GDS and the main output sent on to the filtering section.

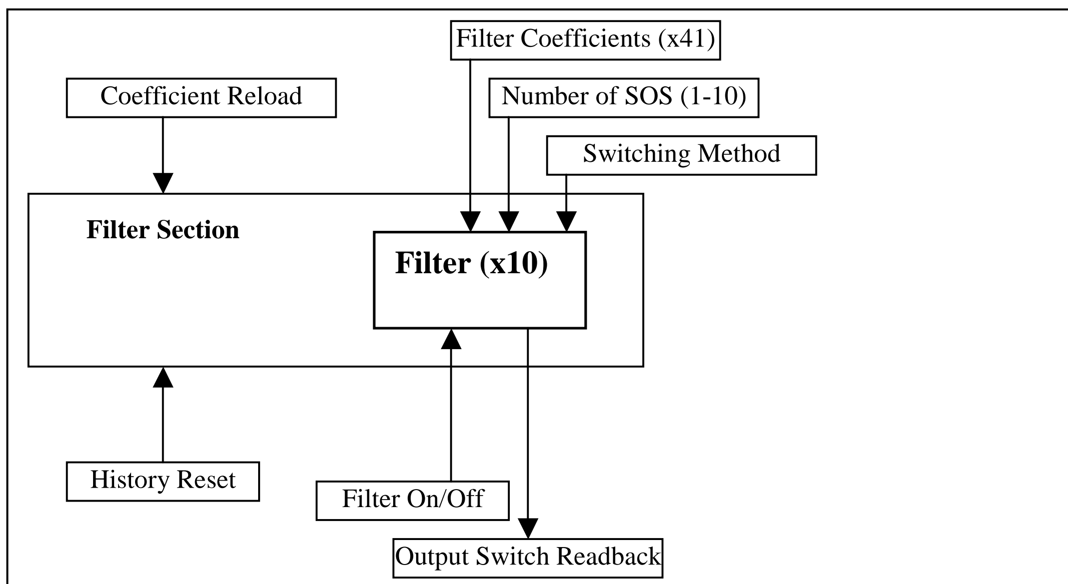


8.1.1.2 Filtering Section

The filter section may have up to 10 IIR filters defined, with up to 10 SOS each. These filters are user definable via a coefficient text file. The software allows for any/all of these filters to be redefined “on the fly” ie a FE process does not need to be rebooted, restarted or otherwise interrupted from its tasks during reconfiguration.

Each filter within a SFM may be individually turned on/off during operation. Various types of switching are available. These switching types are described in detail in a later section.

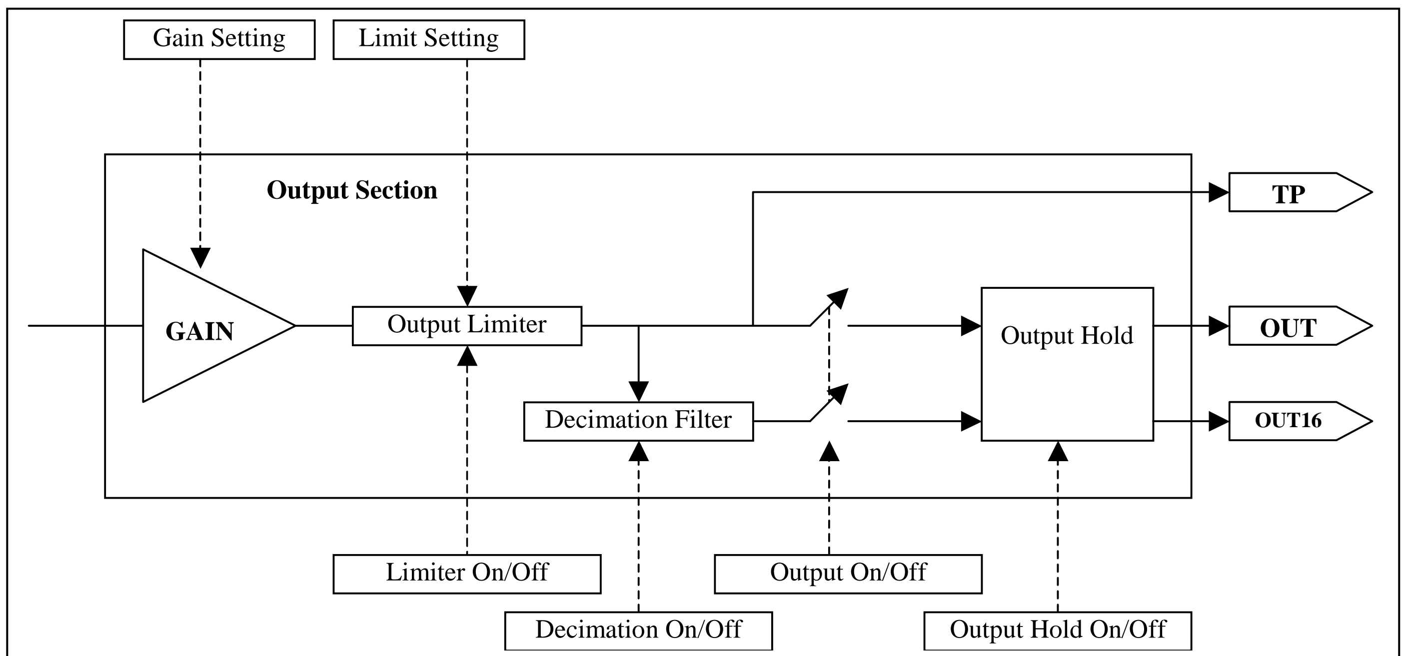
To facilitate implementation of these filter features, a number of control and status signals interface to the filter section. These are shown in the following figure.



8.1.1.3 Output Section

The following figure shows the output section. The output section provides for:

- A variable gain to be applied to the filter section output. This gain may be ramped over time from one setting to another by setting the gain ramp time.
- This output to be limited to a selected value.
- A GDS TP. This TP is always on, regardless of whether the output is turned on or off.
- Ability to turn output on or off.
- A decimation filter to provide a 16Hz output (typically used by EPICS).
- A "hold" output feature. When enabled, the output of the SFM will be held to its present value.

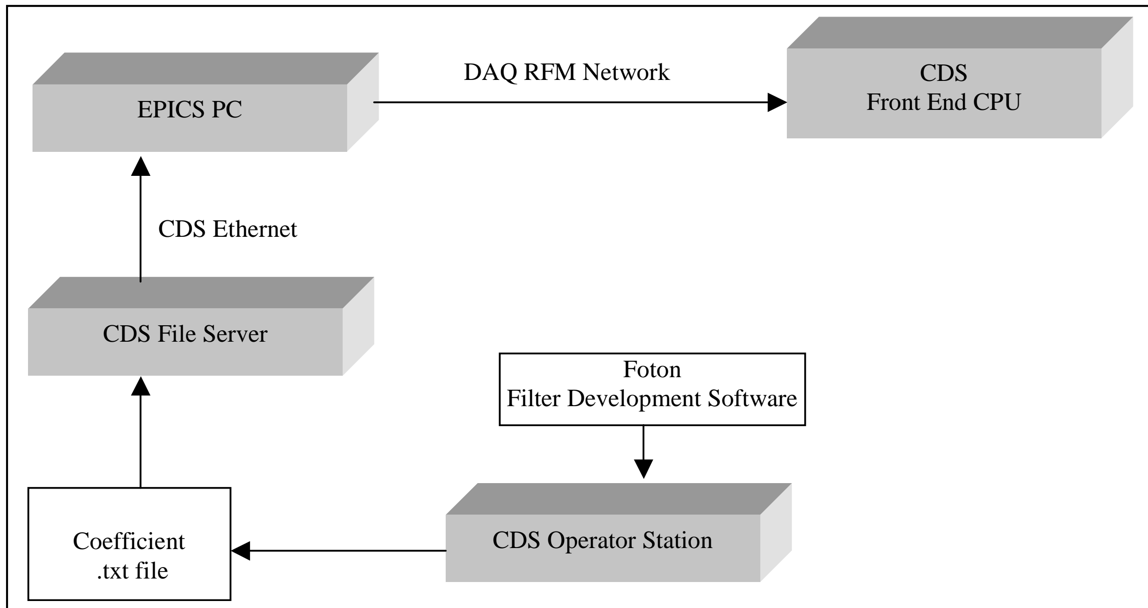


8.1.2 SFM Filter Design and Communications

For SFM to be used in FE processors, a specific hardware architecture must be used and coefficient files generated and downloaded. This architecture is shown in the following figure.

Typically, a GDS tool (*foton*) is used to design the filters for a FE system. This tool generates the filter coefficients in the proper format for use by the SFM. These filters and coefficients are stored in a .txt file, one file defined for each CDS FE CPU.

Once the file is produced, the coefficients are downloaded to the FE processor via an EPICS sequencer running on one of the EPICS PCs. The EPICS PC reads the file via the CDS Ethernet and then relayed to the FE via predefined areas in RFM.



8.1.2.1 Coefficient Files

The filter coefficient files used by the SFM must be located in the */cvs/cds/site/chans* directory. In operational LIGO systems, this file is only to be edited using the GDS *foton* tool. This file contains:

- The names of all SFM defined within a FE processor. Each SFM within a front end is given a unique name in the EPICS sequencer software used to download the SFM coefficients to the front end. These names must be provided in this file for use by *foton*. This is done by listing the SFM names after the keyword 'MODULES'. As an example, from the LSC FE file:
- # MODULES DARM MICH PRC CARM MICH_CORR
- # MODULES BS RM AS1_I
- A line (or lines) for each filter within an SFM, describing filter attributes and coefficients. These line must contain the information listed in the following table, in the exact order given in the table.

Field	Description
SFM Name	The EPICS name of the filter module to which the remaining parameters are to apply.
Filter Number	The number of the filter (0-9) within the given SFM to which the remaining parameters are to apply.
Filter Switching	As previously mentioned, individual filters may have different switching capabilities set. This two digit number describes how the filter is to switch on/off. This number is calculated by input_switch_type x 10 + output switch type. The supported values for input switching are; • 0 – Input is always applied to filter. • 1 – Input switch will switch with output switch. When filter output switch goes to 'OFF', all filter history variables will be set to zero. Four types of output switching are supported. These are: • 0 – Immediate. The output will switch off as soon as commanded. • 1 – Ramp: The output will ramp up over the number of cycles defined by the RAMP field. • 2 – Input Crossing: The output will switch when the filter input and output are within a given value of each other. This value is contained in the RAMP

	field. • 3 – Zero Crossing: The output will switch when the filter input crosses zero.
Number of SOS	This field contains the number of Second Order Sections in this filter.
RAMP	The contents of this field are dependent on the Filter Switching type.
Timeout	For type 2 and 3 filter output switching (input and zero crossing), a timeout value must be provided (in FE cycles). If the output switching requirements are not met within this number of cycles, the output will switch anyway.
Filter Name	This name will be printed to the EPICS displays which have that filter. It is basically a comment field.
Filter Gain	Overall gain term of the filter.
Filter Coefficients	The coefficients which describe the filter design.

8.1.2.2 EPICS Sequencer

To provide for the downloading of filter coefficients to FE processors and to communicate SFM data between the FE processor on the RFM network and operator stations on the CDS Ethernet, an EPICS sequencer must be provided. This sequencer must run on a Linux PC connected to both networks.

In the latest code revisions, this EPICS software and database records are automatically generated by a PERL script. Details of this process follow in a later section of this document.

8.2 SFM Code Details

The SFM software resides in the *cds/rts/src/include/drv/fm10Gen.c* file. The various SFM data structures are defined in *cds/rts/src/include/fm10Gen.h*.

8.2.1 Data Structures

There are three primary data structures associated with SFM. These structures are used to communicate all information into and out of SFM. These structures are:

- VME__COEF, which communicates all of the SFM filter coefficients and operational characteristics, such as switching types, ramps, timeouts, etc. This structure is used to convey the information found in the SFM configuration files to the SFM from the RFM network.
- COEF, which holds all of the runtime filter parameters. This is a subset of VME_COEF, along with the history buffers required for IIR filters.
- FILT_MOD, which communicates all of the dynamic runtime data, such as switch settings, I/O values, etc.

Since it is important to understand these structures to properly use and communicate with SFM in FE software, these structures are further described in the following subsections.

8.2.1.1 VME_COEF

VME_COEFF describes the structure in which SFM coefficient information is transmitted in RFM. It is essentially an array of size MAX_MODULES of the VME_FM_OP_COEF structure. The fields within this latter structure are defined in the following table.

Field	Data Type	Description
filtCoeff[FILTERS][MAX_MODULES]	Double	The IIR filter coefficients for all 10 filters within an SFM.
filtName[FILTERS][32]	char	SFM identifying name (only used by EPICS). Must be unique within a FE system.
bankNum	Int	SFM ID number. Must be unique within a FE

		system.
filtSections[FILTERS]	Int	Number of SOS within each of the 10 SFM filters.
sType[FILTERS]	Int	Type of switching to be used for each of 10 filters.
ramp[FILTERS]	Int	With type 2 output switching, the number of cycles over which to ramp up/down a filter output. For type 3 switching, the +/- range for comparison switching.
timout	Int	If type 3 or 4 output switching, the time to wait for switching conditions to be met before finally switching by default.
Crc	Unsigned int	CRC checksum calculation for coefficient validation.

8.2.1.2 COEF

The COEF structure is an array of size MAX_MODULES of the FM_OP_COEF structure. This structure is used by SFM to internally keep track of and calculate SFM data. This data structure is defined in the following table.

Field	Data Type	Description
filtCoeff[FILTERS][MAX_COEFFS]	Double	The IIR filter coefficients for all 10 filters within an SFM
filtHist[FILTERS][MAX_HISTRY]	Double	History buffers for all 10 filters within an SFM
filtSections[FILTERS]	Int	Number of SOS per filter for all 10 filters within an SFM.
sType[FILTERS]	Int	Switch type settings for SFM filters.
decHist[8]	Double	History buffers for 16Hz decimated output of SFM.

8.2.1.3 FILT_MOD

This structure defines fields that are continuously (typically 16Hz) sent to/ read from EPICS via the RFM network and a few field which are necessary for runtime SFM computations. This structure is composed of two other structures, FM_OP_IN and FM_OP_OUT. These structures are defined in the following tables. Those fields which the FE software must send to EPICS at 16Hz are denoted by an *. Those which must be received from EPICS at the same rate are denoted by **.

8.2.1.3.1 FM_OP_IN Structure

Field	Data Type	Description
opSwitchE**	UINT32	Control/Status word sent from EPICS for each SFM
opSwitchP*	UINT32	Control/Status word sent to EPICS from each SFM
rset**	UINT32	History and Coefficient reload request flag.
offset**	Float	Offset to be applied to SFM input.
outgain**	Float	Gain to be applied at output of SFM.
limiter **	Float	+/- limit to be applied to SFM output.
rmpcmp[FILTERS]	Int	The ramp or comparator value to use for filter switching.
timeout[FILTERS]	Int	Timeout value for comparison and zero crossing switching filters.
cnt[FILTERS]	Int	Keeps track of present ramp count for ramping filters.
gain_ramp_time	Float	Gain change ramping time in seconds

The opSwitchE and opSwitchP variables provide SFM control and status information. This is done through the definition of the 32 bits contained in these variable. The control/status register bits are defined in the following table.

Bit	Name	Description
0	Coeff Reset	This is a momentary bit. When set, the EPICS CPU will read in new SFM coeffs from file and send this information on the FE via the RFM network. The FE SFM will read and load new filter coefficients from RFM.
1	Master Reset	Momentary, when set, SFM will reset all filter history buffers.
2	Input On/.Off	Enables/disables signal input to SFM.
3	Offset Switch	Enables/disable application of SFM input offset value.
Even bits 4-22	Filter Request	Set to one when an SFM filter is requested ON, or zero when SFM filter requested OFF.
Odd bits 5-23	Filter Status	Set to one by SFM when an SFM filter is ON, or zero when SFM filter is OFF.
24	Limiter Switch	Enables/disables application of SFM output limit value.
25	Decimation Switch	Enables/Disables application of decimation filter to SFM OUT16 calculation.
26	Output Switch	Enables/Disables SFM output (SFM OUT and OUT16 variables)
27	Hold Output	If (!bit 26 && bit27), SFM OUT will be held at last value.
28	Gain Ramp	If set, gain of filter module != requested gain. This bit is set when SFM gain is ramping to a new gain request.

8.2.1.3.2 FM_OP_DATA Structure

Field	Data Type	Description
filterInput **	Float	Input signal to SFM
exciteInput **	Float	The GDS EXC signal applied to this SFM.
inputTestPoint	Float	Sum of filterInput and exciteInput.
testpoint **	Float	SFM output testpoint.
output **	Float	Output of the SFM.
output16Hz **	Float	Decimated output of the SFM.

8.2.2 Required Headers and Source Files

Like the other standard CDS software modules, the SFM code is intended to be included into and compiled with the individual FE software. This requires the fm10Gen.c file and specific headers to be included into the global declarations of the FE software. These are:

```
#define MAX_FILTERS      /* Total number of IIR filters in FE system = SFM x 10 */
#define MAX_MODULES  /* Total number of SFM in FE system */

#include "fm10Gen.h"        /* SFM data structure definitions. This header will also include fm10Gen.c */
#include "iscNetDsc.h"      /* This header will define where SFM information for all FE is located in
                                the RFM network. This file must be updated to provide space for new
                               systems */
#include "gdsLib.h"          /* This file provides the GDS TP ranges for all FE systems. These TP numbers
                               are required to properly map the SFM TP */
#include "daqmap.h"          /* The file provides the DAQ RFM mapping information required to send
                               requested SFM TP to DAQ */
```

#include “drv/daqLib.c”   /* Besides providing the routines to write data to DAQ, this file contains the rfm_refresh() routine required to update SFM data during initialization. */

8.2.3 SFM Usage in FE Code

8.2.3.1 Initialization

On startup, the FE software must get all of the SFM information from the EPICS PC via the RFM network. This requires two calls to *rfm_refresh()*, one to ujpdate the VME_COEF structure, and one to update the operator settings found in the FILT_MOD structure. This is done by first setting a pointer to the RFM memory base address, using the findRfmCard() function, then offsetting to the specific FE SFM structure locations in the RFM_FE_COMMS structure, defined in iscNetDsc.h.

Once the data has been updated to the local RFM module, this data is copied into local memory. The FE code must then call the *initVars()* routine, which will initialize all of the SFM parameters. The SFM are now ready for use by the FE software.

8.2.3.2 Runtime Calls to SFM Software Routines

During run time, there are a number of calls available to SFM routines. The first three described below are presently used in site installed software. However, two new routines are now available (*filterModule2()* and *filterModuleD()*, which will soon replace the first three.

- inputModule(FILT_MOD **pFilt,** / * Pointer to FE code defined SFM structure */
  int **modNum** /* The identying SFM number */
  ):

This software will perform all of the functions associated with the SFM input section, as previously described.

- filterModuleId(
  FILT_MOD ***pFilt,** /* Filter module data pointer */
  COEF ***pC,** /* Filter coefficient pointer */
  int **modNum**, /* SFM ID number */
  double **filterInput** /* Input data sample (output from inputModule() */
  int **id** /* For systems using multi-dimensioned SFM arrays */
)

This routine performs the tasks described for the filtering and output sections of the SFM.

- filterModule(
  FILT_MOD ***pFilt,** /* Filter module data pointer */
  COEF ***pC,** /* Filter coefficient pointer */
  int **modNum**, /* SFM ID number */
  double **filterInput** /* Input data sample (output from inputModule() */
)

This routine simply calls filterModuleId, with the id variable set to zero. This allowed legacy code to use the filterModuleId function, which was developed later.

- filterModule2 (
  FILT_MOD ***pFilt,** /* Filter module data pointer */
  COEF ***pC,** /* Filter coefficient pointer */
  int **modNum**, /* SFM ID number */
  double **filterInput** /* Input data sample (output from inputModule() */
  int **mask** /* Allows external switching control */
)

In certain applications, it is required to switch filters within different filter modules synchronously. This filter module call provides that capability. This filter module routine was recently updated to incorporate

the functions provided by *the inputModule()* routine. Therefore, software using this routine should not call *inputModule()* first (as is required by *filterModuleId()* and *filterModule()*.

- filterModuleD(

```
FILT_MOD *pFilt,      /* Filter module data pointer */
COEF *pC,             /* Filter coefficient pointer */
int modNum,           /* SFM ID number */
double filterInput    /* Input data sample (output from inputModule() */
int id                /* For systems using multi-dimensioned SFM arrays */
)
```

This is the latest filter module routine, and should be used in place of the combination *inputModule(), filterModuleId(), filterModule()* calls in all new code development and as existing software is updated. Like *filterModule2()*, it now has the *inputModule()* software incorporated within this function. Also, unlike the earlier functions, this filter module routine and *filterModule2()* both handle inputs and outputs as doubles. The earlier calls cast inputs and outputs as floats.

8.2.3.3 Data Communications To/From EPICS

The communication and update of SFM data to/from EPICS is the responsibility of the specific FE software. Typically this information is updated for all SFM at 16Hz. The SFM structure fields which must be updated by the FE software are defined in the FILT_MOD structure definition tables (section 6.2.1.3) and denoted with an * or **. Examples of how present FE software provides for EPICS communications may be found in source files under *cds/rts/src/fe*.

The SFM software does have one additional call, which should be made by the FE software at the same update rate (typically 16Hz) as the EPICS channels. This is a call to checkFiltReset(). This call will check for new filter coefficients or requests for clearing history buffers. If a SFM has been sent new coefficients, this routine will read the coefficients from RFM, verify that they have been received correctly, by performing a CRC checksum test, and then load the new coefficients into the proper SFM. Other than making a call to this routine, the specific FE software does not need to deal directly with filter coefficients and their updates.

8.3 Code Compilation

All CDS FE source code should reside in *cds/rts/src/fe*. A Makefile exists in this area for compiling all FE source code, with various compiler flags set. The only compiler flag required to use SFM code is either –DSERVO2K or –DSERVO16K. This defines to the SFM software at what rate it will be running.

9 DAQ/GDS Software

All CDS realtime front end processors must have the capability to send data to the DAQ system and to receive diagnostic excitation signals from the GDS AWG. Data rates are from 256Hz to 16384Hz in 2^nHz steps. In order to better understand the GDS/DAC software described later, a brief description of the DAQ system is provided in the following subsection.

9.1 Overview

The basic architecture of the DAQ/GDS system and the interconnections to FE processors is shown in the following figure. The key hardware components are:

- CDS File Server: Hosts the DAQ/GDS channel configuration files.
- DAQ FrameBuilder: Writes data to disk in LIGO standard frame file formats and delivers data on request to operator stations via a Network Data Service (NDS), or off site, via an NDS Proxy.
- TPM/AWG: This GDS device selects GDS TP/EXC signals and sends GDS EXC signals to CDS FE on request.
- EPICS PC: Conveys DAQ channel configurations to FE processors.
- RFM network (2 per IFO) to communicate DAQ/GDS data and configurations in realtime.

9.1.1 GDS and DAQ Files

Any data channel defined within a front end processor's code as a "GDS Test Point (TP)" may be acquired. These TP are automatically defined for CDS Standard Filter Modules (SFM), described in section 6 of this document. The individual front end code may also define additional TP channels beyond those defined by SFM. Which channels are to be acquired and stored to disk is dynamically configurable without interrupting and/or restarting any of the front end systems.

All TP within all CDS front ends must be assigned a unique TP name and number and listed in a GDS parameter file. There is one GDS parameter file per interferometer, which is downloaded to the GDS TPM/AWG processor. This processor will allow selectionj/deselection of GDS signals via RPC calls on the CDS Ethernet.

The GDS .par file contains the following information for each TP:

Parameter	Example	Description
Signal Name	[H1:LSC-DARM_EXC]	Name of the channel. This must be all caps and follow the LIGO naming convention.
Interferometer number	ifoid = 1	Site interferometer to which this signal is assigned (1 or 2)
RFM Number	rmid = 1	Which RFM network contains this data channel.
DCU ID	dcuid = 13	Each front end is assigned a unique ID number. This field defines from which front end this signal originates.
TP Number	chnnum = 1	This is the unique TP number for that data channel.
Data Type	datatype = 4	Several data types are available. However, by default, all TP within the GDS param file must be of type float (4).
Data Rate	datarate = 16384	Though data may be acquired and stored at various rates, the rate listed in this file must be the full data rate of the front end which is sending this data channel (2048 or 16384).

Placing the data information in the GDS parameter files only provides a description of channels to the DAQ and GDS systems. This allows all TP to be located and assigned by the DAQ/GDS system (see description of GDS code in next section), but not acquired or saved to disk. Which channels are to be saved to disk is defined by individual files for each front end. These are files with the .ini extension found in the */cvs/cds/lho/chans/daq* directory at LHO and in the */cvs/cds/llo/chans/daq* directory at Livingston. These files contain the following information:

Parameter	Example	Description
Channel Name	[default] [L1:LSC-AS_I]	The unique channel name for this data channel. All further parameters listed will apply to this channel. Typically, [default] is listed first in the file to set the following parameters for all channels listed, unless a new parameter is given. Individual channel names **MAY NOT** be the same as listed in the GDS parameter files for the same TP number.
TP number	chnnum=10152	This is the TP number associated with these data channel. It needs to be the same as one of the TP listed in the GDS parameter file. However, as noted above, it must have a unique and different name than the name given in the GDS parameter file to this TP number.
Front end ID number	dcuid=17	Again, each front end is assigned a unique ID number, same number as in GDS parameter file.
Acquisition data rate	Datarate=16384	This number may be between 256 and 16384, in multiples of two. It may not be faster than the front end processor rate given for that same TP listed in the GDS parameter file.
Gain selection	gain=1.0	Front ends which only do data acquisition contain ICS110b modules with programmable gain. For such systems, the gain setting is described

		in this field. This field is ignored in systems which do not have this capability.
Acquire Flag	acquire=1	DAQ is to store this channel to disk (1) or just provide realtime data access to this channel (0).
IFO ID number	Ifoid=1	Which of up to two IFO at each site the data channel is associated with.
Data Type	datatype=4	A data channel may be acquired and stored as several different data types, with 16 bit short and 32 bit float the most common.
Units	Units=V	The engineering units of this signal.
Slope and Offset	Slope=6.102e4 Offset=0	The slope and offset to be used to convert the raw data signal to engineering units. All data is stored as raw data. These fields and the units field are stored separately in the data frames for later converstion as desired.

Once the .ini file has been configured and or modified to the desired settings, the operator initiates an upload of the DAQ parameters via an EPICS screen. This causes a sequencer on an EPICS Linux PC to read the .ini file and write a subset of the parameters into the DATA INFO BLOCK for that particular front end on the RFM network and then notify the front end that a new DAQ configuration exists. The front end will then configure itself with this new information and data acquisition will begin.

9.1.2 RFM Network Layout

All communication of DAQ/GDS channel information and data is done over CDS RFM networks. The space allocation of this memory is described in the following table and defined in *cds/rts/src/include/daqmap.h*.

Address Range	Space Designator	Description
0x00000000 – 0x01ff0000	Controls Area	Reserved for communications between front end processors and front end processors to EPICS PCs.
0x01ff0000 – 0x02000000	Data Info Block	Contains the information that the front ends require to send data to the DAQ system, including signal numbers and rates.
0x02000000 – 0x02040000	IPC Area	Contains status information passed between front ends and the DAQ system.
0x02040000 – 0x02100000	GDS Control Block	This area is used to select GDS TP and EXC signals.
0x02100000 – 0x03f00000	DAQ Data Blocks	Memory space where front ends write data to the DAQ system. Each front end is assigned a 1Mbyte block of memory to write one second of data.

Basically, the FE CPU will read in its Data Info Block and check the GDS Control Block to determine which data channels it is to write to DAQ or which GDS EXC signals it is to read in and apply. The FE DAQ status is communicated to the DAQ via the Inter-Process Communication (IPC) area.

The data from a front end is sent via a 1Mbyte memory block, a separate such block defined for each front end CPU. This memory block is subdivided into 16 sections, each containing 1/16 second of data. As a FE CPU completes writing 1/16 second of data, it also computes and sends a CRC check sum for that block. The DAQ FrameBuilder will pull off 1/16 second of data from every FE approximately 1/8 second later, compute and compare the CRC check sums, and store the all of the data into a LIGO frame in local memory. Every 16 seconds, the DAQ FrameBuilder will then write a data file. With this arrangement, each FE CPU is limited to 1Mbyte/second acquisition rates and up to 128 channels.

GDS TP and EXC signals are assigned their own 1Mbyte RFM blocks. There are a total of four such blocks, one each for 2048Hz TP and EXC signals and one each for 16384HZ signals. The GDS AWG is the only CPU to write into the EXC blocks. Data into these blocks is written at least 1/8 second in advance, such that the signals are ready to be read by the FE CPU processors at the proper time. The TP areas are written to by all FE CPUs, which interleave their data into these memory blocks based on the TP selection and the TP order in the GDS Control Block.

9.2 Data Acquisition Software Library

The software to handle data acquisition for all front end processors is located in the *cds/rts/src/drv/daqLib.c* file. There is a single routine (*daqWrite()*) which must be called once during front end code initialization and then again at the end of every front end cycle (2048Hz or 16384Hz).

9.2.1 Software Functional Description

9.2.1.1 Calling Parameters

The *daqWrite()* routine requires the passing of the following parameters from the calling FE code:

- Int flag: Set to zero when called during system initialization. Thereafter, must be set to one when called at the end of each front end compute cycle.
- Int dcuId: The unique ID number of this front end on its DAQ RFM network.
- DAQ_RANGE daqRange: This structure defines the valid TP ranges associated with this front end. These ranges are unique to a front end and defined within front end header files.
- Int rfmType: The type of RFM module to be used for data acquisition (VMIC5565_NET or VMIC5579_NET).
- Int rfmNum: Which PMC slot the DAQ RFM is located in. Many front ends are connected to multiple RFM networks, only one of which connects to the DAQ system.
- Int sysRate: This is the raw acquisition rate of the front end for 1/16 second (DAQ_16K_SAMPLE_SIZE or DAQ_2K_SAMPLE_SIZE). This code needs to know how fast the front end code is running to properly write data to the RFM network.
- Float *pFloatData[]: An array of pointers to front end TP not associated with SFM.
- FILT_MOD *dpsPtr[]: An array of pointers to the front end code SFM.

9.2.1.2 Initialization Tasks

Prior to the writing of DAQ data, the daqWrite() task must initialize various parameters and pointers. This initialization sequence is as follows:

1. Create two data buffers in local memory, each buffer to store 1/16 second of data. These will be used as swing buffers, with data written to one while the other is read from to send data to RFM.
2. Set up pointers to these two buffers.
3. Find the base address of the RFM module to be used for DAQ and reset the FAIL light on the front panel.
4. Set up pointers to the DAQ IPC and Data Info Blocks in the RFM network for this particular front end.
5. Request an update of the RFM memory for this front end. This is done by a call to rfm_refresh(). Since the front end was probably off line prior to code start, the data in its RFM memory is most likely stale. Therefore, it must send a request to the EPICS processor to update the DAQ configuration information prior to proceeding with initialization.
6. From the DAQ INFO BLOCK area, get the DAQ configuration file CRC checksum information. This value will be periodically sent to the DAQ Framebuilders via the front end DAQ IPC area. This allows the Framebuilders to verify that the front end is running with the same channel configuration as they expect.
7. From the DAQ INFO BLOCK area, read in the DAQ channel count. The code then verifies that the channels to be acquired does not exceed a maximum limit (presently 128 channels max / front end.) This channel count is also relayed to the Framebuilders via the IPC area.
8. From the DAQ INFO BLOCK area, read in all necessary channel information. This includes the TP number, data type and data rate. This information is used here to calculate the size (in bytes) for 1/16

second of data. This size is passed to the Framebuilders through the dataBlockSize variable in the IPC area.

9. Calculate the amount of data (in bytes) to be transferred to the RFM network on each front end cycle to complete the full transfer of a 1/16 second block of data. This is dependent on the total bytes for 1/16 second of data and the cycle rate of the front end code. To facilitate DMA to the RFM network, this number will always be a multiple of 8 bytes. Because of this, this code may not write data to the RFM on every cycle. The data block sizes and transfer byte count information are printed out for inspection.
10. The area in RFM where the data is written is divided into 16 equal areas, one for each 1/16 second of data. Therefore, for all front ends, 1 second of data exists in the RFM network at all times. The Framebuilders will extract data in 1/16 second blocks. The code will now set up pointers to these 16 memory blocks in RFM.
11. Develop a local lookup table for front end data. Given the configuration information, the code now sets up pointers to the front end code TP as they exist in local memory. This later allows the code to copy TP from front end code memory into its DAQ swing buffers in the proper format.
12. If the RFM module is a VMIC5565, the daqWrite() code may use DMA to transfer data out of the swing buffers to the RFM. In order to do this, the DMA registers on the RFM module are located and a pointer to this area initialized.

9.2.1.3 DAQ Runtime Software

After each control cycle, the specific FE software must call *the daqWrite()* routine with the 'flag' parameter set to DAQ_WRITE. The routine has to be called each cycle in order for the daqWrite() to properly maintain its pointers and other information.

On every call, this routine will perform the following functions:

- If this cycle is the first of a one second period and a reconfiguration command has been received (reconfig = 1 in FE CPU assigned Data Info Block), the code will begin a DAQ configuration setup. This routine is similar to the initialization routine. Since it would take too long to run this configuration all within the time period of one control cycle, the code will perform one channel of configuration on each call for the next 1/16 second or until complete. During this 1/16 second, data acquisition from this FE will be stopped. In this fashion, DAQ channels from FE CPU may be reconfigured (via the .ini files) without interruption of the FE CPU control functions.
- If a reconfiguration is not in progress, write out data.
    - For each DAQ channel configured, get the data from the pFloatData[] * or dspMod[] * location, based on the local lookup table built during the initialization process. This data is loaded into a local short or float data variable, depending on the DAQ data type selection. This data is then written into one of the local swing buffers in the proper location for that channel and the particular control cycle. The local swing buffer mimics how data appears on the RFM network in the data block areas, that is 1/16 second of channel 1, followed by 1/16 second of channel two, and so on.
    - From the swing buffer written for the previous 1/16 second by the routine above, write out data to the RFM network. During initialization, the number of bytes to write each cycle was determined. Note that while the initialization routine attempts to equalize the number of bytes written on each cycle, it must also maintain an 8 byte boundary. This latter condition typically results in a higher bytes/cycle result, which means that 1/16 second of data transmission will complete before all the controls cycles (128 for 2048Hz or 1024 for 16384Hz) run. Therefore, once the last of the data for a 1/16 second block are completed, this code will not continue to write data until the next 1/16 second period begins.
    - Perform a CRC check sum calculation on the data sent to RFM. On each cycle, this is only done on the bytes transmitted to RFM on that cycle. Note that waiting for 1/16 second of data to be transmitted and then perform the CRC check sum calculation on that entire block would take too long to process. Therefore, just a piece of the final calculation is done on each cycle.

Once a complete 1/16 second block of data has been transmitted, the DAQ code must send some additional information to validate the data and do some housekeeping to set up for the next 1/16 second block:

- Send the .ini configuration file CRC checksum via the ipc->fileCrc location to the FrameBuilders. This allows the FrameBuilders to verify that the FE CPU is using the same channel configuration information as they are.
- Send a cycle count (0-15) via the ipc->cycle location. This indicates which 1/16 second of data was just written and is used by other DAQ software to verify that this FE CPU is synchronized with GPS.
- Complete the CRC checksum calculation for the entire 1/16 second data block and transmit it via the ipc->bp[].crc location. When the DAQ FrameBuilders read in this 1/16 second data block, they will perform their own CRC calculation to verity that data was not corrupted during the transmission process.
- Switch local swing buffers. The buffer from which data was being transmitted to RFM now becomes the local data buffer and the local data buffer filled during the previous 1/16 second now becomes the data transmission buffer.
- Check for DAQ reconfiguration command in Data Info Block for this FE CPU.
-

9.2.2 Code Usage

9.2.2.1 Prerequisites

Each node on the DAQ network must have a unique number and name associated with it. If a new system is to be added, this file must be updated.

9.2.2.2 Includes

The *daqLib.c* file is intended to be included into specific FE software and compiled as a single object with that software. The code is not intended to be compiled stand alone. Therefore, within the global declarations area of the FE code, the following lines must be included:

```
#include "daqmap.h"   /* Includes DAQ pointer and variable definitions */
#include "gdsLib.h"       /* Defines the GDS TP ranges for all FE CPUs */
#include "drv/daqLib.c"   /* Includes the daqWrite() routine */
```

9.2.2.3 Routine Calls

During the initialization phase of the FE software, the daqWrite() routine must be called with the ‘flag’ parameter set to “DAQ_CONNECT”. Prior to this call, the FE software is responsible for setting up the following parameters to be passed to daqWrite():

- dcuId: This is the unique id number of a FE CPU for a particular interferometer. This number must be defined in the daqmap.h file. This number will determine where daqWrite() will read/write data from/to the DAQ RFM network.
- daqRange: This structure includes the TP number ranges which are valid for this FE CPU. These ranges must be defined in gdsLib.h. As DAQ and GDS channels are selected, daqWrite() will use this information to determine if these channels are associated with this FE.
- *pFloatData[]: This is where the daqWrite() routine will look to find TP data not associated with SFM. This must be in the same order as the TP are listed in the GDS .par file.
- *dspPtr[]: This array of points to the SFM modules within the FE code. This is where daqWrite() will look for data associated with SFM.
-

9.3 Global Diagnostics

9.3.1 Overview

The DAQ system and the daqLib software within the FE processors allowed for the passing of selected FE data channels to the DAQ FrameBuilders, primarily for data archiving. While these channels are runtime configurable, the DAQ .ini files are semi-permanent. That is to say that the channels selected are those that are most useful in the later offline analysis and searches for gravitational waves and other LIGO studies. Therefore, these channel lists do not change very often. However, FE processors typically have many more data channels available which are interesting during specific commissioning, testing and interferometer performance studies. It is not desired that these channels be permanently saved, but rather to be available on request to be sent to LIGO data viewing and analysis tools, such as the DAQ Dataviewer or GDS Diagnostic Test Tools (DTT). This data becomes available to the FrameBuilders on a temporary basis and may be distributed by the DAQ NDS in the same fashion as permanent DAQ channels.

For testing and interferometer calibration, it is also useful to inject arbitrary waveforms into FE systems. This is analogous to injecting test signals into electronic hardware using a signal or function generator. This capability, plus the ability to get TP data from FE processors, falls under the category of GDS.

9.3.2 Functional Description

The DAQ/GDS overview discussion of section 7.1 provides a system architectural overview and some of the supporting configuration files required and RFM network layout. To handle all of a FE processor's tasks associated with GDS, including checking TP/EXC selections and reading/writing GDS data from/to RFM, a standard software routine was developed. This routine is called once during initialization and thereafter once after each FE control cycle. The source code for this routine is contained in *cds/rts/src/drv/gdsLib.c*.

During initialization, this function sets up the GDS pointers into the RFM network and initializes a local lookup table. This local table will be used to store information required from the GDS Control Block as well as pointers to the TP/EXC data within local memory.

During runtime, this function will write selected TP channel data to the RFM network, one value per TP per cycle. All TP are written as floating point numbers, regardless of their native data type. This function will also read in selected EXC signals, one value per EXC per cycle. Along with verifying that an EXC number is valid for this FE, the code will also check the "data valid" field for each EXC channel each time it reads in a value. If the data valid field. If the data valid flag is zero, the code will set the EXC signal to zero.

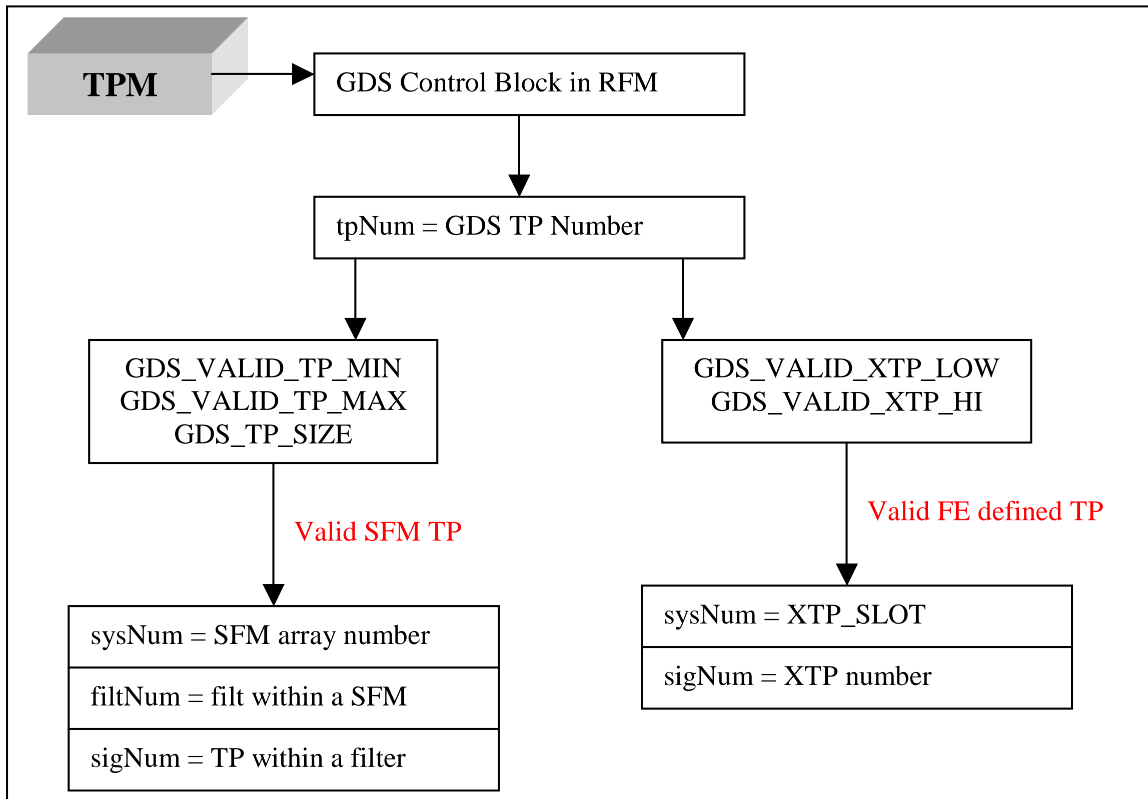
9.3.2.1 Test Points

TP are dynamically selectable and the GDS TP table updated once per second by the TPM. The FE software must check this table once per second and enable any valid test points and remove deselected TP. The FE software checks this table within the time frame of the 15th of the sixteen 1/16 second DAQ data blocks.

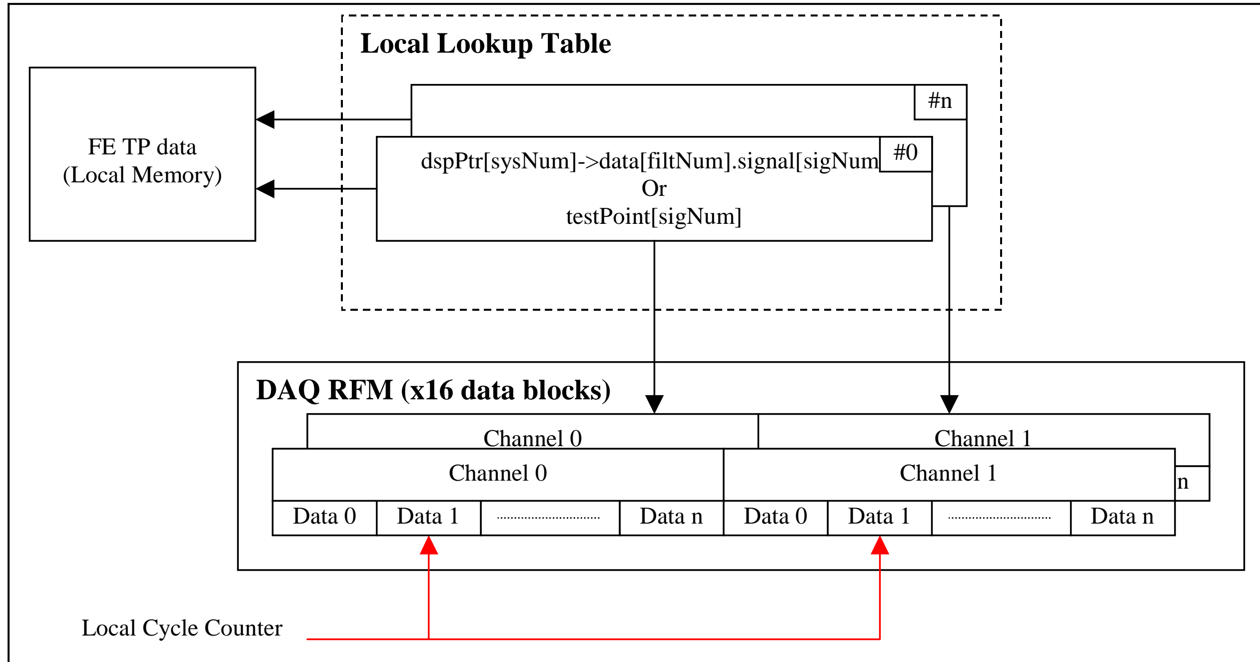
The process of enabling GDS TP and decoding GDS TP data is basically as described in the following text and figure.:

1. A TP is selected by its unique name using one of the DAQ or GDS tools.
2. The DAQ or GDS tool requests the TP channel from the DAQ FrameBuilder
3. The FrameBuilder, in turn, requests that the selected TP be enabled via an RPC call to the Test Point Manager (TPM) via the CDS Ethernet. The TPM runs on the GDS AWG processor.
4. The TPM looks up the TP number for this TP name (as defined in the GDS .par file).
5. The TPM places this TP number into the next available space in the GDS Control Block area of the RFM network. The TPM updates this table in RFM once per second.

6. The FE processor checks the GDS Control Block once per second and places the TP number into its local lookup table (tpNum field).
7. The FE software then checks to determine if this is a TP number to which this FE should respond. It does this by comparing against the TP ranges for this FE. If the selected TP is valid for a FE SFM TP, then it will fall in the range of >= GDS_VALID_TP_MIN and <GDS_VALID_TP_MAX. If it is a valid TP not associated with an SFM, then the TP number will fall within the range of >=GDS_VALID_XTP_LOW and < GDS_VALID_XTP_HI. These ranges for all FE processors must be defined in the *gdsLib.h* file in *cds/rts/src/include*.
8. If the TP selection is within the defined range, the FE code will load the remainder of its local lookup table for that TP slot. This information will point the TP writing code to the TP data in local memory and, by its position in the table, which TP channel the data is to be written to in RFM.
9. If the TP is not valid for this FE, the software will denote this by placing a –1 in the sysNum field in its local table.



Once a TP has been selected, it is written by the FE processor every control cycle, as depicted in the following figure and described below.



When called, the FE software loops through its local lookup table. If the sysNum field is not –1, this FE must write data to the corresponding channel in RFM. The sysNum, filtNum and sigNum variables in the table then point to the data passed to the routine in local memory. This data must now be passed on to RFM.

Within the RFM networks, two TP data blocks are assigned for writing TP channel data, one for 16384Hz data and one for 2048Hz data. Each data block is 1Mbyte in size, again divided into sixteen 1/16 second blocks, which in turn are subdivided into a predefined number of channels. Within a channel are 128 or 1024 data points, determined by the rate of 2048Hz or 16384Hz. The channel number is determined by the TP location within the local lookup table array. Which of the sixteen data blocks to write to and the data point within the channel are determined by a local cycle counter, which is calculated each time the code is called. Once the correct position in RFM is located, the local TP is written to RFM, always in floating point data format.

9.3.2.2 Excitation Channels

The layout and selections for GDS EXC signals is similar to TP. Two 1Mbyte data blocks are assigned in RFM for EXC signals, one for 2048Hz channels and one for 16384Hz channels. The process of enabling EXC signals is basically as follows:

1. A GDS EXC signal is selected by name using one of the GDS software tools. Using these tools, the type of waveform to be generated, frequencies, amplitudes, etc. are selected.
2. The particular tool is commanded by the user to begin sending the EXC signal. The EXC parameters are conveyed to the TPM and AWG.
3. The TPM sets the EXC number in the GDS Control Block during a one second period and the AWG begins writing the waveform to the RFM on the following 1 second boundary.
4. The FE processor checks the GDS Control Block for an EXC number which is valid at some point after the TPM has set the EXC number. If an EXC channel is valid for the particular front end, it will begin reading in and applying this signal starting on the next 1Hz boundary.

Again, the position of the EXC number within the GDS Control Block table identifies where within the 1Mbyte EXC RFM memory the particular channel data is to be read from. As with TPs, the local cycle counter tracks which 1/16 data block should be read from and which data point within a channel should be

read. Unlike TP, this data pointer is set one cycle ahead. Therefore, it reads in the EXC data that is needed by the FE application a cycle in advance.

9.3.3 Code Usage

9.3.3.1 Prerequisites

Prior to using the gdsLib.c software, there are some prerequisites to be met. First, the gdsLib.h file must be updated to reflect the GDS ranges for a particular FE processor. The following TP/EXC numbering standard applies to GDS signals and must be followed:

00000 – 10000	Range for 16384Hz EXC channels.
10000 – 20000	Range for 16384Hz TP channels.
20000 – 30000	Range for 2048Hz EXC channels
30000 – 40000	Range for 2048Hz TP channels
40000 -	Reserved for GDS DAC channels

Within this file, the following parameters must be defined for each FE processor. First are five parameters to define the ranges of valid EXC numbers for this FE application. The EX_MIN and EX_MAX define the range of valid EXC numbers associated with the FE application. Since applications may use multi-dimensional arrays of SFM, EXC_SIZE must be set to indicate to the gdsLib software how large the lower array element is in order for it to increment the next dimension of the SFM array. For example, for purposes of GDS, the code expects that the SFM pointer is sfm *[ ][EXC_SIZE]. The XEX_LOW and XEX_HI values determine the range of available FE EXC channels which are not associated with SFM.

```
#define GDS_VALID_EX_MIN          /* Lowest value of valid EXC numbers associated w/SFM */
#define GDS_VALID_EX_MAX          /* Highest value of valid EXC number */
#define GDS_EXC_SIZE              /* SFM array size – sfm *[ ][EXC_SIZE] */
#define GDS_VALID_XEX_LOW       /* Lowest value of valid EXC number not associated w/SFM */
#define GDS_VALID_XEX_HI        /* Highest value of valid EXC number not associated w/SFM */
```

A similar set of five parameters exist for GDS TP, as follows.

```
#define GDS_ETMX_VALID_TP_MIN
#define GDS_ETMX_VALID_TP_MAX
#define GDS_ETMX_TP_SIZE
#define GDS_ETMX_VALID_XTP_LOW
#define GDS_ETMX_VALID_XTP_HI
```

Once the GDS ranges have been defined in the gdsLib.h header file, the FE software may be completed. To operate with the other CDS systems, the GDS TP/EXE information must be added to the GDS .par file as described previously. Whenever this GDS .par file is modified, the GDS TPM and DAQ FrameBuilders must be reset before the new listing takes effect.

9.3.3.2 Includes

In order to make use of this code, the specific FE application must include the following in the global declarations area.

```
#define MAX_MODULES          /* Need to know number of SFM within this FE */

#include “fm10Gen.h”         /* Needed to find SFM TP */
#include “iscNetDsc.h”       /* RFM network definitions */
#include “daqmap.h”          /* Defines location of GDS Control and data memory */
#include “gdsLib.h”          /* Defines valid GDS TP/EXC numbers, ranges */
```

```
#include “drv/gdsLib.c                    /* Include the GDS source code */
```

9.3.3.3 Routine Calls

Within the FE code, the GDS software is called by the following:

```
gdsUpdate(
        int initFlag,          /* 0 = Initialize code,
                                  1 = Update TP signals only
                                  2 = Update EXC signals only
                                  3 = Update both TP and EXC signals */
        int daqNet,            /* Type of RFM module used to connect to RFM network, either 5565 or 5579.
                                  In either case, code expects to use first module of that type found on CPU. */
        FILT_MOD dspPtr[],       / * Array of pointers to SFM data */
        float testpoint[],       /* Array of pointers to TP data not associated with SFM */
        float excSignal[],       /* Array of pointers to EXC data channels not associated with SFM */
        int gdsMonitor[][]       /* Structure for returning selected TP/EXC numbers to the calling code
        )
```

Prior to calling this routine during initialization, it is the responsibility of the specific FE application to set up the array of pointers to SFM and additional TP/EXC signals. During runtime, the initFlag variable may be set to 1, 2 or 3. Typically, it is set to 3, but may be set to 1 or 2 if the FE code desires to write TP somewhere else within its code than where it needs to read EXC signals.

9.3.3.4 Compilation

The FE code should be located in the cds/rts/src/fe directory and use the Makefile for compilation. The only compiler flag specifically required by gdsLib.c is the FE rate (-DSERVO2K or –DSERVO16K).

10 Building EPICS Systems

As previously mentioned, EPICS systems which support realtime front end processors are run on Linux PCs. These PCs are presently 3.0GHz Pentium processors in a 2U high rack mount configuration. Each PC contains an RFM module to interface to its various front end processors.

For each interferometer, there are three such PCs, each supporting a group of realtime processors. These three units are:

- susepics: Epics for the corner station suspension controllers, connecting via a VMIC5565 RFM network.
- iscepics: Interface to the LSC and ASC front ends via a VMIC5565 RFM network. Both iscepics and susepics reside on the same RFM network.
- dcuepics/hepiepics: This PC maintains Epics for the end station suspension controllers and the ADCUs. At LLO, this unit also runs Epics for the Hepi system. The connection to front ends is via a VMIC5579 RFM network.

For communications between Epics processors and front end processors, there are typically three memory areas defined and allocated for each front end, or group of front ends performing like tasks, such as Hepi processors. These areas are:

- Front end specific Epics channels: Each front end typically has unique information which must be passed to/from Epics. A structure is defined and set aside for this purpose.
- SFM Data: Data to/from SFM, less the SFM coefficients.
- SFM Coeff: Coefficients for the SFM.

The structures which define the particular front end Epics communications and RFM pointers to all three Epics communications structures are maintained in the *iscNetDsc.h* file.

The Epics sources and databases are maintained in *cds/rts/src/epics* subdirectories.

10.1 Automatic code and database generation

10.1.1 Perl Script

For most systems, Epics sequencers and databases are generated in part via a perl script. This script is maintained in *cds/rts/src/epics/util/fmseq.pl*. The *util* directory also contains *skeleton.st* and *skeleton.db* files for use by the perl script. This script will build all of the Epics sequencers and database records associated with CDS filter modules, user defined matrices, and user defined Epics I/O records. This script inputs user defined files in the cds/rts/src/epics/fmseq directory to build the associated Epics.

10.1.2 Input Files

Presently, there are perl script input files for generating Epics for all but the suspension Epics. These files have a standard format which must be followed to properly generate the Epics sequencers and databases. These files must be placed in the *cds/rts/src/epics/fmseq* directory.

- List of Filter Modules: File must contain a list of names for all of the filter modules, each on a separate line. This list should not contain site or system specific prefixes or SFM standard suffixes.
- Following the filter module list, there must exist a line, beginning with the keyword EPICS, which describes the data structure for passing EPICS data to the associated front end processors. After the keyword EPICS, this line must contain the following items, as defined in iscNetDsc.h:
    - Name of the Epics data structure.
    - Pointer to the SFM data in RFM.
    - Pointer to the SFM coefficients in RFM
    - Pointer to the EPICS data structure in RFM.

As an example from the hepi file:
EPICS HEPI_EPICS hepidsp.hepiDsp[sysnum] hepicoeff.hepiCeoff[sysnum] hepiepics.

- List any matrices, one per line. Note, at present, these matrices may only be inputs from the operator (Epics) to the front end. Each line must start with the keyword MATRIX, followed by an Epics record name, size, and assoiciated Epics RFM variable. For example:

  MATRIX STS_MTRX_ 8x4 comms[sysnum].sts_matrix

  Note that when the Epics db records are built for matrices, the numbering will start at 11. For example, the line above will produce STS_MTRX_11, STS_MTRX_12, etc.

- Individual Epics records may be added by using the keywords INVARIABLE or OUTVARIABLE at the beginning of lines. INVARIABLE indicates a value to be passed from Epics to the front end and OUTVARIABLE a value from the front end to Epics. As an example:

  INVARIABLE RAMP_TIME comms[sysnum].rampTime int ai 1 field(HOPR,"65000")

  - INVARIABLE: indicates that this value is to be passed from Epics to the front end.
  - RAMP_TIME: the name of the Epics database record to be created, less the standard prefix.
  - comms[sysnum].rampTime: the variable defined within the RFM network for passing this data, as defined in the iscNetDsc.h file.
  - int: indicates the data type (may also be float).
  - ai: indicates the Epics record type. This may be any supported Epics record type.
  - 1: Value to be assigned to this record at Epics startup.
  - field: List any additional fields to be produced with the Epics record, such as PREC, etc.

- System prefixes for EPICS db names needs to be added with a line that begins with the 'systems' keyword. This allows the script to assign the proper EPICS name prefix to the names given previously for the filters and other in /out variables. An example for ASC would be 'systems ASC-'. The script can also build multiple systems from the same file by listing multiple entries on this line. For example, multiple etm systems are created from the same file by listing 'systems SUS-ETMX_ SUS-ETMY_.

- The script will automatically produce a gds.param file for the filter modules. In order to do this, a line must be defined with the key word 'gds_config'. This keyword must be followed by four parameters. As an example, the line may be 'gds-config 24000 34400 400 600. The four parameters must be as defined in the *include/gdsLib.h* file as follows:

  - Minimum valid GDS excitation number: This is provided in the header file as *system*_VALID_EX_MIN.
  - Minimum valid GDS testpoint number. This is provided in the header file as *system*_VALID_TP_MIN.
  - The maximum number of excitation points per system. When multiple systems are built from the same fmseq file, the script must know the offset between systems. Therefore, this parameter should be the *system*_EXC_SIZE given in the header file. The script will number all filters in the file for the first system starting at EX_MIN. The second system filters will start at EX_MIN + EXC_SIZE, and so on for all systems listed on the *systems* line.
  - The maximum number of test points per system. This is the TP_SIZE given in the header file and is used in the same fashion as excitation numbering of multiple systems, only for test points.

Note that the gds param files generated by this script during the make process will end up in the *rts/build/system* directory. Since it is only the param file for the particular system being built, it must still be incorporated into the site gds.par file by hand.

10.2 Epics Sequencers (.st files)

The Epics records and sequencers which are generated automatically are limited in their capabilities. For example, INVARIABLE and OUTVARIABLE defined in the script files simply allows for the communication of data between the front ends and Epics ie no other operations may be done on the data by Epics, such as calculations. Therefore, separate Epics sequencers must be written when more than just the straight passing of data must be performed. These .st files are maintained in cds/rts/src/epics/seq directories.

10.3 Epics Database (.db) Files

Whenever unique .st files are created, as in 9.2 above, the user must create the associated database records manually in .db files. These files are maintained in cds/rts/src/epics/db directories.

10.4 Epics Build

A Makefile exists in the *cds/rts/config* directory for each Epics processor. The make command itself must be run in the cds/rts directory. The Makefiles are setup such that the resulting Epics objects and db files are generated for all three interferometers in the *cds/rts/target* directory, along with a startup script and autoburt file. Several key items must be included in the Makefile:

- TARGET = :This will define both the name of the Epics executable file to be generated and the name of the target subdirectory in the cds/rts/target directory.
- SRC = (+=): Names of all the .st files to be compiled into the final TARGET executable. For custom .st files, this would be of the form SRC+= src/epics/seq/mySequencer.st. For state code auto generated by the perl script, this would be of the form SRC += build/$(TARGET)/fmseqFile.st. Adding .c files to the build is done in the same fashion, such as SRC += src/drv/rfm.c.
- DB += : List of .db files to be included in the build.
- IFO = : Indicates which interferometers to build code for, such as IFO = H1 H2 L1, which would build code, databases, startup commands, and autoburt files for all three interferometers.
- SEQ += : The command for starting up a sequencer and arguments to be passed to it, such as SEQ += ‘daqConfig,(“ifo=%IFO%,site=%SITE%,sys=SOS,sysnum=19”)’.
- Compile and library options for the appropriate RFM network type. For example, for code to run on a PC connected to a VMIC5565 net, the following two lines must be included:
    - CFLAGS += -DRFM_5565
    - LIBFLAGS += -L/usr/lib/rfm2g –lrfm2g
- Extra make commands. This is usually a list of sed replace commands for building multiple databases from the same auto code generation file. For example, the cds/rts/src/epics/fmseq/hepi file will generate a generic database which must be modified to be unique for each chamber hepi controls. Therefore, the Makefile.hepiepics has some Extras defined, such as:

```
build/$(TARGET)/hmc1.db: build/$(TARGET)/hepi.db
    sed ‘s/%IFO%/L1/g;s/%SYS%/SEI-MC1_/g’ $< > $@
```

11 Software Installation

All CDS realtime and supporting EPICS software is to be installed in subdirectories of the */cvs/cds/site/target* directory at the observatories. These subdirectories are to be named in accordance with site standards and done so in coordination with the site software engineer.

As a guideline for installation, the following check list is provided.

1. Ensure all new software has been properly placed and/or updated in the CDS CVS repository. Also make sure that all of the latest CVS updates are included in the build directories.
2. Compile the latest software.
3. Perform a code backup if new code is replacing existing software.
4. Install the realtime front end code into the appropriate target subdirectory. Ensure a *startup.cmd* file either exists or is created and verify that it is correct.
5. Retrieve the gds.par file generated by the EPICS compilation and merge into the site gds.par file. This file should be checked that proper TP numbers and ranges were generated and there are no conflicts.
6. Add any extra GDS excitation or test points to the site gds.par file ie those not automatically generated during the EPICS compilation.
7. Ensure that there is a proper data acquisition .ini file in the */cvs/cds/site/chans/daq* directory to support this front end code.
8. Check the /cvs/cds/site/chans/daq/master file to verify this front end's .ini file is listed. This is the file the framebuilder uses to get DAQ channel lists.
9. Ensure that there exists at minimum a basic digital filter coefficient file for this front end code in the */cvs/cds/site/chans* directory. In particular, ensure that all proper filter module names appear in the header of this file. If filter module names have changed, ensure all old names are either modified to be correct with latest software or removed.
10. Install the supporting EPICS software, including all of its subdirectories, in the appropriate target area. Ensure that the autoburt file is correct and has been moved up to the target area.
11. Install any new or updated EPICS screens to the appropriate */cvs/cds/site/medm* subdirectory.
12. Ensure that the site 'Red Book' gets updated with all system reboot parameters.
13. Make a site elog entry specifying what software has been updated and a brief description of the updates and changes.