Available on CMS information server **CMS IN -2009/003**

CMS Internal Note

The content of this note is intended for CMS internal use and distribution only

12 December 2008

Control Software for the CSC Track Finder

D. Acosta, J. Gartner, D. Holmes, K. Kotov, A. Madorsky, L. Uvarov, B. Scurlock, H. Stoeck, D. Wang

University of Florida, Gainesville, Florida, USA

F. Geurts

Fermi National Accelerator Laboratory, Batavia, Illinois, USA

M. Matveev

Rice University, Houston, Texas, USA

Abstract

This note describes the online control software for the CMS CSC Track Finder. It details the core library which interacts with the hardware through HAL and the CAEN controller driver. On top of the core library lie several libraries containing generic hardware manipulating functions. These range from standard board initializations to self tests and pattern injection tests. Wrapping the functions libraries are the control interfaces; stand alone for local CSC subdetector control and a CSCTF Trigger Supervisor cell for control coming from the central CMS trigger. This note contains details on the design of the modules and code examples intended to demonstrate common implementation methods pertinent to new developers.

1 Introduction to the CSC Track Finder

The CMS CSC Track Finder is described in [1] and its functionality within the CMS level-one trigger in [2] and [3]. The installation and commissioning of the crate is described in [4] and [5] respectively.

The basic function of the Track Finder is to receive CSC trigger primitives originating from individual CSC chambers and to try to extrapolate combinations of them into candidate muon tracks. Through fast identification of IP pointing and high Pt muons, the Track Finder represents the first stage in the pipelined trigger and cuts the raw LHC rate of up to 40 MHz down to (a tuneable) 10s of KHz. The L1 candidate muons found by the CSC Track Finder are passed on to the Global Muon Trigger (GMT) which serves to combine inputs from all 3 CMS muon subsystems. The DT (Barrel) and CSC (End Cap) Track Finders exchange trigger primitives coming from the overlap region in their respective subdetectors. This allows implementation of functionality to prevent a reduction in trigger efficiency at the end cap - barrel boundaries. The task of receiving CSC trigger primitives and ranking potential muon tracks must be done within a strict latency budget of around half a microsecond.

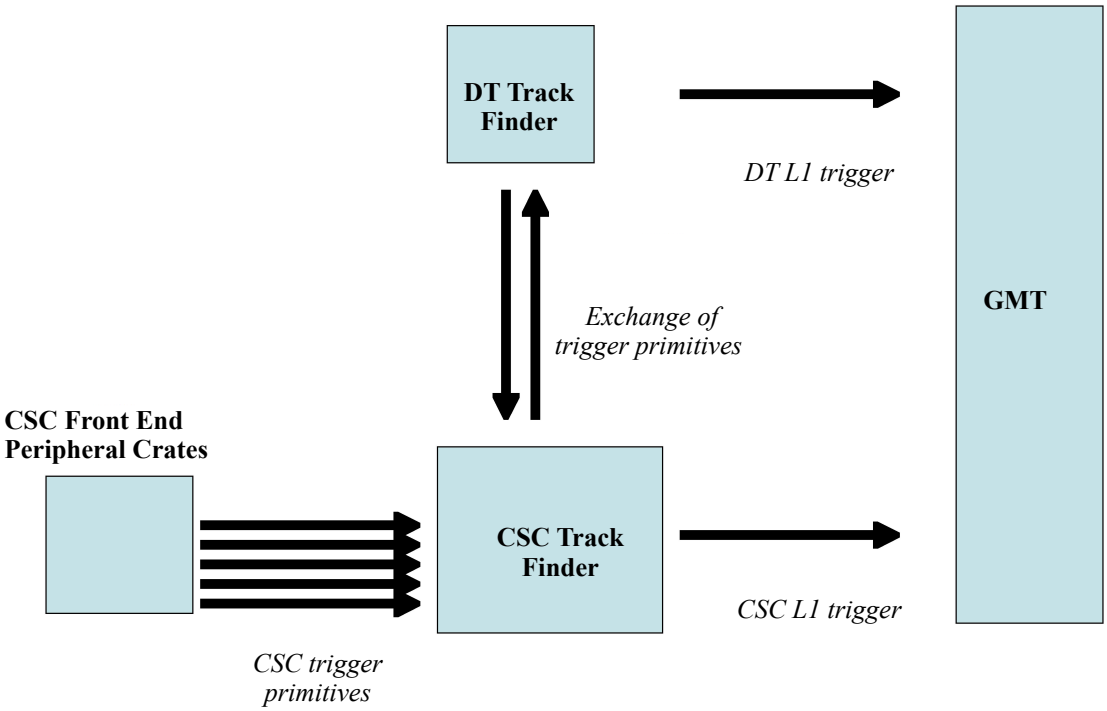

Figure 1. An overview of the CSC Track Finder within the CMS L1 Trigger Path.

The CSC Track Finder consists of a single crate containing 16 9U boards. The guts of the work is done by twelve Sector Processor (SP) boards. Each of the SPs receives CSC trigger primitives from chambers in a single End Cap, within a region subtended by a 60^{o} slice in phi and produces up to 3 corresponding candidate muon tracks per 25 nS bunch crossing. Six of the SPs cover one CSC End Cap and 6 cover the other. All twelve of the SPs pass their candidate muon tracks on to the single Muon Sorter (MS) board. The MS then filters the (up to 36) incoming candidate tracks down to a maximum of 4 candidates per bunch crossing and passes these on to the GMT which resides in the Global Trigger (GT) crate. The other boards in the crate are the Clock and Control board (CCB); a generic CSC board for distributing the TTC clock and fast commands over the backplanes of CSC crates, the Device Dependent Unit (DDU) which is responsible for collating the trigger data and passing it on to global DAQ via SLINK and CSC local DAQ via optical fiber and the CAEN VME controller card through which one is able to address the other boards.

2 Overview of the Track Finder control software

The Track Finder control software described in this note refers to that used to control the Sector Processors (SP), the Muon Sorter (MS) and the Clock and Control board (CCB). The Device Dependent Unit (DDU) used to stream trigger data to DAQ is controlled through the same CAEN VME controller[12] but by the CSC FED Software package which is not described here.

The control software package is mostly written C++ and is compiled within and dependent upon the XDAQ[11] framework. The code itself is stored in the CERN CMS TriDAS CVS[9] repository. The path to the base directory is TriDAS/trigger/csctf. The platform is CERN Scientific Linux[14] and the package, framework, machine and architecture supports multithreaded core machines.[1]

The control software is expected to be used in 2 basic modes; firstly within the CMS trigger architecture during CMS and CSC running of the main Track Finder Crate, and secondly standalone, on potentially non-standard test beds. The basic structure of the package is split in to 3 levels;

The base level refers to a single dynamically bound library that is responsible for linking to the proprietary CAEN[12] and SBS[13] VME drivers via the XDAQ package HAL. The most fundamental object is the SPobject which can be used to make reads and writes to registers in FPGAs on a given board. It is intended that one SPobject should be instantiated for each board with which there is to be communication. The package exports an interface which includes a factory method for instantiating and initializing SPobjects as well as a class to register and manage all of the SPobjects corresponding to the 14 boards in a single container object.

The functions library level refers to a set of libraries that contain standard functions and manipulations pertinent to the various boards. These range from basic initialization to Lookup Table loading and test pattern injection. Typically one passes a pointer to an SPobject to a function in the functions libraries and the function then makes a series of writes and reads to the board via the pointer. The functions libraries all include interfaces that can be implemented from any framework or standalone code as well as a directory of standalone test executables directly exporting the most important functions of the library.

The last tier of the control software is the Trigger Supervisor[15] package. The CSC Track Finder Trigger Supervisor Cell is the link between the CMS Trigger Supervisor Central Cell and the Track Finder base and functions libraries. CMS Run Control[17] parses messages to the Trigger Supervisor Central Cell which then relays them to the trigger subsystem local cells via SOAP[16]. The cells respond to the commands and then return a SOAP message back up the chain. The Trigger Supervisor group provide the class templates for the subsystem cells and then the subsystem developers implement the appropriate code within the local cells to make their hardware perform in the required manner. The local cell further provides a web based interface allowing the commands that would come from the central cell during CMS running to be replicated during local CSC running.

[1] At the time of writing (November 2008), the framework required a 32-bit Linux operating system but the upgrade to 64-bit SLC is expected to be a trivial one and should happen in 2009.

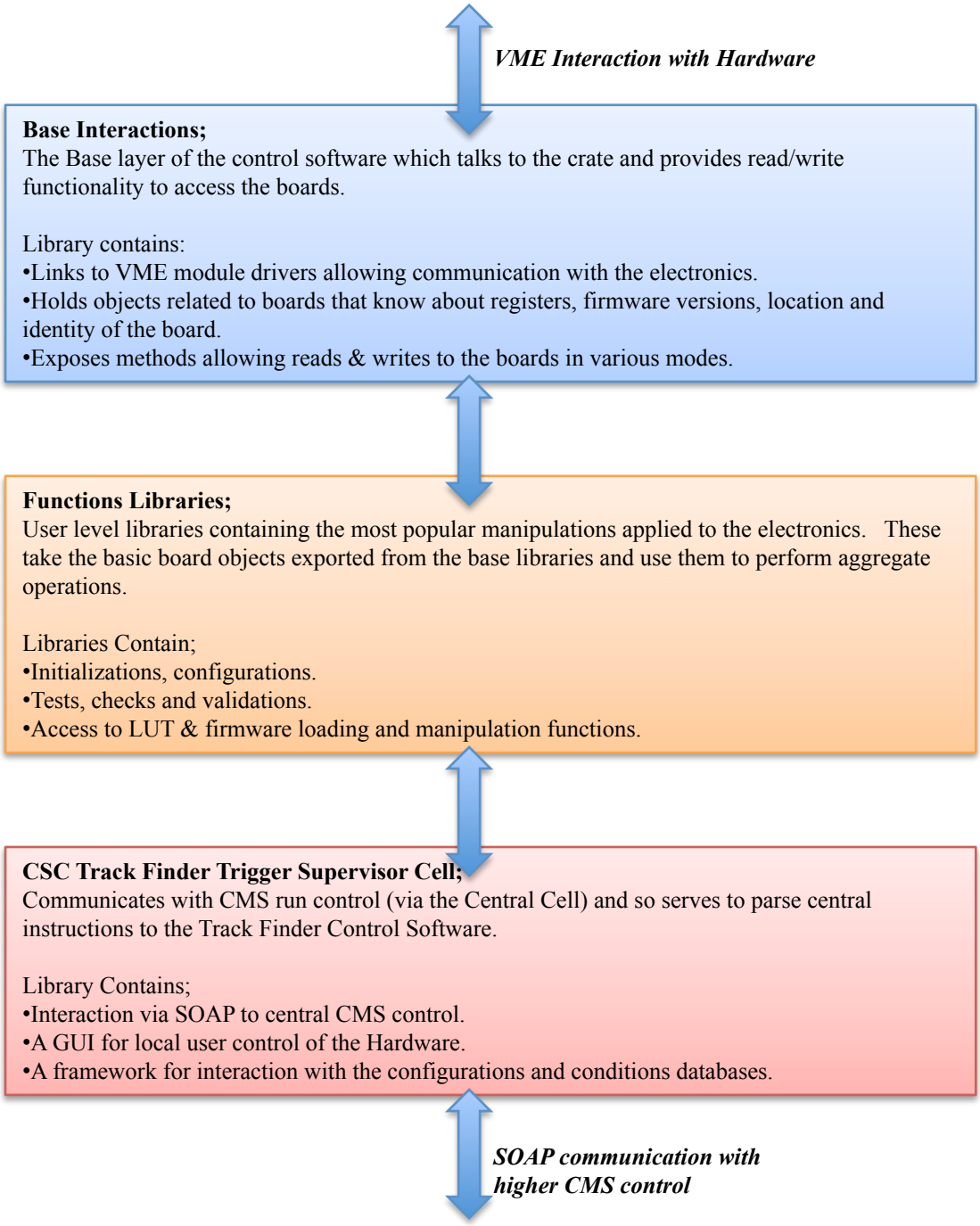

Figure 2. The 3 basic tiers of Track Finder Control Software

The exact configurations to be loaded in to the hardware are stored in a set of Oracle[18] object relational 'configuration' databases. The Trigger Supervisor Central Cell parses a key to the local Track Finder Cell which then queries the database for the corresponding settings to be loaded to the hardware. The local cell is similarly a conduit for the conditions (slow controls) data read from the hardware and sent on to a 'conditions' Oracle database. The electronics is typically queried periodically with a configurable frequency of the order of tens of seconds to minutes.

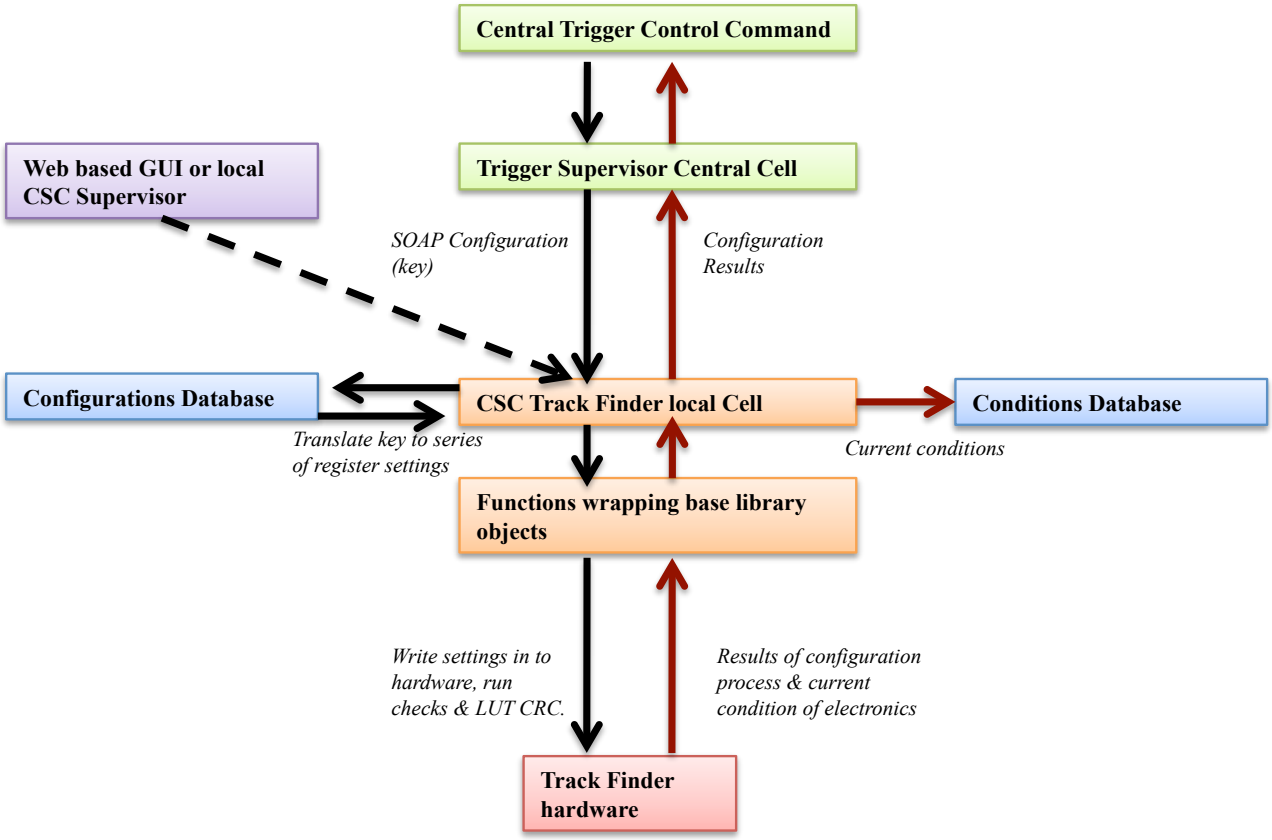

Figure 3. The Flow Diagram for a Standard Trigger Configuration.

Figure 3 demonstrates the basic configuration sequence. The configuration command, in the form of a SOAP message containing a configuration key, is initiated by either Central CMS Control (global running), the CSC Supervisor (local running) or the web based GUI(Track Finder stand alone tests). The SOAP command is received by the Track Finder Control PC and the key is decoded. The Track Finder Software receives the key, queries the configurations database for the corresponding electronics register settings and then writes them in to the hardware. Various checks are performed on the success of the configuration sequence and a response with the result is parsed back up the chain to the progenitor. The electronics is periodically pulsed for its current state and the results are filled in to the conditions database.

During compilation, one must follow the order of first compiling XDAQ and the VME interface drivers, then next the SPBaseInteractions library which depends on various XDAQ packages. After that, one may compile the functions packages detailed in Section 3.3 onwards, which depend on the SPBaseInteractions objects. Finally one may compile the Trigger Supervisor cell described in Section 4 which has dependencies ranging across all of the CSCTF software. An example compiling process is described in Appendix 3.

3 Introduction to the packages & principal software objects

3.1 Structural Overview

The code itself resides in the CSC Track Finder directory in the CMS software *TriDAS* tree. Specifically the base directory is *TriDAS/trigger/csctf*. If the Track Finder environment is correctly set up then an environmental variable, *$CSCTF,* will point to the *csctf* base directory. Inside */csctf/* are several base packages which are described in this section. Each of the subdirectories follows a similar layout structure. Source code is inside */src/* directories with corresponding header files in */include/* directories. One can compile the source code in a package through the *Buildfile* in the base directory. Running it causes a shared object library (.so) to be compiled and placed in the */lib/* directory. Code to wrap and implement the package member functions lives in */test/* directories along with a makefile which will produce compiled executables in the */bin/* directory.

Code is archived using CERN Concurrent Version System (CVS) [9]. CVS provides a repository where code can undergo structured release. The code is accessible via a browser to the CVS repository linked from [9]. Automatic pseudo-UML documentation of the code is done through DOXYGEN[10]. DOXYGEN is simple to re-run on each significant new code release and provides systematic description of the code functionality and structure. The DOXYGEN script along with an instructional *readMe.txt* resides in *csctf/doxygen/*. Output of the automatic documentation is linked from the group website [10]

3.2 SPBaseInteractions

The SPBaseInteractions package builds the library that contains the core software objects responsible for direct communication with the Track Finder Crate. The objects contained within it depend only on libraries available through XDAQ[11] and as such is the first of the CSCTF packages to be compiled. Most of the other CSCTF packages make use of and depend upon the core objects supplied.

The object central to CSCTF software is the ‘SPobject’. One SPobject is instantiated for each electronics board (12 SPs, a CCB and a MS) and it allows the user to directly read and write registers on that board. Contained within the SPobject is the map of registers translating to given base address offsets, the firmware dates of the corresponding board and the VME bus adapter through which the crate is accessed. The SPobject itself is actually a composite object containing objects for each of the FPGAs and it is technically these objects that allow reads and writes to the boards.

Typically users get hold of SPobjects through use of an ‘SPObjectParser’ object. This is a class which exists to generate SPObject pointers as the user demands them. It contains sensible defaults for parameters such as the register base address offset map, the VME bus adapter and the expected firmware dates for the various boards. As the SPObjectParser instantiates a new SPobject, it makes several basic communication, clock lock and firmware date checks before parsing the requested object to the user.

Users interacting with a whole crate of TF Boards rather than just a single one at a time should use a ‘TFCrateContainer’ object to construct and store their SPobjects. This container object allows the user to hold a single pointer through which they can access all of their boards. It incorporates several instantiation modes depending on the level of automation, detection and sensitivity to errors the user requires as it constructs an interface to all the boards in the crate.

It is important to note that one actually deals in pointers to SPobjects. SPobject* is what is returned to the user via the SPobjectParser or the TFCrateContainer and is what is expected as an argument by the functions libraries. Copying (and hence parsing) SPobjects by value or by assignment is actually prohibited by the mechanism of inheritance of the class *SPBaseInteractions/include/unCopyable*.[2] Similar is true for the FPGAObject, SPObjectParser and TFCrateContainer objects.

[2] This is an important proofing feature and is necessary for complex composite objects such as the ones we use here in the absence of explicitly defined copy constructors and assignment operators.

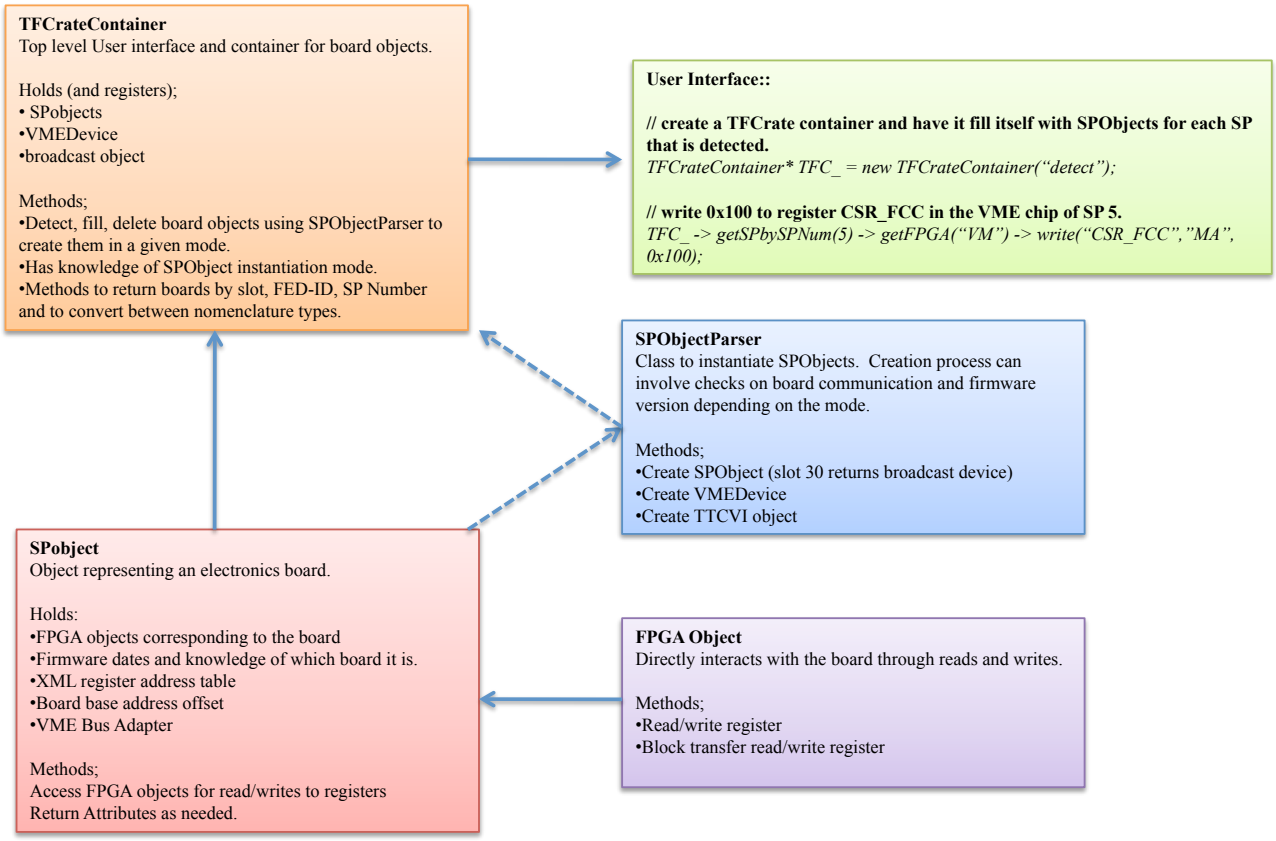

Figure 4. A diagram of the basic interactions of the base library classes and the way in which one makes a simple write to a Sector Processor.

Upon instantiation, one can parse one of several initialization modes to the TFCrateContainer or SPObjectParser which will then dictate the settings with which the SPobjects are built. The SPobjects can be queried for the mode setting with which they were constructed. If one does not parse a mode then the default mode is used. Table 1, below, gives an explanation of the various modes of instantiation.

Table 1. The modes in which the base objects can be instantiated. If no mode is parsed then default is used.

Mode	Appropriate object	Translation
no arg ()	TrackFinderCrateContainer	This is the same as parsing *full*.
empty	TrackFinderCrateContainer	No boards filled, an empty container is returned.
full	TrackFinderCrateContainer	All boards will be filled in to the container.
detect	TrackFinderCrateContainer	All boards that respond to a handshake will be instantiated. Those that do not will be ignored.
dummy	TrackFinderCrateContainer or SPObjectParser	All boards filled but instantiation using dummy bus adapters. This means no actual hardware access takes place. It allows software to be tested independently of the hardware.
debug	TrackFinderCrateContainer or SPObjectParser	All warnings and errors during instantiation are displayed but ignored. Use with caution.

Mode	Appropriate object	Translation
SBS	SPObjectParser	SPobjects are built using an SBS bus adapter appropriate to an SBS VME controller. See Appendix 4.

Further details on re-implementing an SBS bus adapter are given in Appendix 4.

Example code showing how to instantiate the basic objects and then use them to perform Track Finder tasks using the functions libraries is given in Appendix 1.

The SPBaseInteractions package also contains classes called *FWChecks* and *utilities*. Although neither of these represent functionality related to interaction with the hardware, they are depended on by classes that do and hence they need to be in the first of the compiled libraries. Both are exported outside the library and have functions that can be implemented at any level in the code hierarchy. FWChecks deals with reading, interpretation and verification of firmware versions loaded to boards in the Track Finder Crate. It is automatically used for verification purposes as SPobjects are built and can be called for immediate reads interactively through the *SPFunctions/test/readCrate_FWnConfig.exe* executable. The *utilities* class is just that, it is a variety pack of C++ functions that are called regularly from all over the online software packages. It includes functions related to file stream manipulations, generate auto time stamps and do various STL and generic data type manipulations.

3.3 Basic board functions; the SPFunctions package

The SPFunctions library contains common functions applied to the Track Finder Crate boards.

The CCB, MS and MPC have appropriately named classes which contain standard functions specific to that board. These functions range from basic configurations to gets and sets of board parameters and modes, test pattern injections and validating read outs.

The rest of the classes pertain to SP functionality. On the whole, the names give away the area of functionality. Classes exist to configure the SP from scratch and to change the trigger mode. There is a class for getting and setting the various timings and one for reading and testing the FMM STTS responses. There are classes to perform manipulations on LUTs and to work out the real world meaning of the ETA window settings. There is a class to look after the "monitorable" items in the Track Finder which involve all of the slow control readings. There is a command parser allowing streams of commands to be sent to the crate boards and a couple of miscellaneous function classes containing methods to make injections of files of data in to the hardware, corresponding methods for data read out over VME and artificial triggers.

The methods contained in these classes are explained briefly in this subsection but for a more full understanding one should refer to the interface files and the automatic documentation.

3.3.1 Classes for the CCB, the MS and the MPC

For these 3 boards, the production was RICE University and the functions were designed and implemented by M. Mateveev. The code in the classes in the SPFunctions package is simply an implementation of the instructions provided in the board user manuals linked from [8].

CCBfunctions; This class contains functions related to the Clock and Control Board (CCB). The first 2 functions are *CCB01Enable* and *CCB04Enable*. These are simply functions that configure the CCB (pre production 2001 version and production "2004" version) in to default run mode. Most of the rest of the functions in this class refer to interactions with the TTC-RX chip mounted on the TTC-RQ mezzanine board. This chip implements a coarse (25 ns) and fine (2 ns) additive delay to the input of the TTC signals to the chip and hence also to the crate to which the signals are distributed. These delays are set so as to normalize the TTC distribution to all of the Peripheral Crates. For the case of the Track Finder Crate, adjustment of the TTC-RX delay would serve to offset the Track Finder with respect to the rest of the system. As of November 2008, no RX tuning at the Track Finder had taken place.

MPCfunctions; This class has functions related to the Muon Port Card. The Track Finder Crate does not implement a MPC as standard but it does have a test slot where one can be inserted for intra-crate trigger link testing. This class provides functions for standard MPC initialization and reading of basic attributes such as firmware. It also provides a handle to switch in to "sorter mode" where one can uniquely pipe the output from a given TMB to given MPC links.

Lastly there are functions allowing loading of test patterns in to the MPC memory for subsequent injection in to the trigger path.

msConfigs; This is the class relating to Muon Sorter (MS) functions. The functions in this class are almost all direct translations of the MS User Manual [8]. They include basic configuration with standard loading of the LUT, access to the Digital Clock Managers, simple self-tests and several more complex pattern reception and injection trigger chain tests.

3.3.2 Classes relating to SP functionality

The rest of the classes contained in *SPFunctions* relate to Sector Processor (SP) functions.

monitorables; This class contains functions related to the SP "monitorable items" or slow control readout. The crate is typically pulsed every minute or so and basic counters and statuses are read out to be written to the conditions database and displayed on a monitoring panel. The functions in this class provide interfaces to the various counters and status indicators allowing the return of values currently in the hardware.

sTTS_states; This class allows direct setting and read back of all of the SP Synchronous Trigger Throttling States (STTS) as well as functions to scroll through the states. These functions are implemented during tests of the STTS system and the corresponding Track Finder connections to the Fast Merger Module (FMM). Further details on ths interface are found in the Track Finder Installation document in [4].

trigModes; This class allows direct setting of the basic trigger modes of the SP. It includes several "singles" trigger options on a station by station basis as well as more standard coincidence mode settings.

timings; This class allows direct setting and read back of the SP Alignment FIFO and Pipeline FIFO delay timings. There are also functions which can be used to calculate the optimized settings.[3]

initialConfigurations; This class contains functions for stand alone configuring SPs by hand. There are various modes and settings with which one can initialize a SP and they are conveyed through the function nomenclature. There are also methods allowing parsing of a given configuration key or indeed the return of the currently loaded one.

commandParser; This class allows direct parsing of files full of configurations to the SP.

miscSPfunctions; This class contains miscellaneous functions applicable to the SP. Functions include a method to set up fake triggers and tracks, one to read out the contents of the DAQ FIFO and one to dump a stream of data over VME.

SPYFIFOFunctions; This class contains functions related to injection and read out of SP pattern data. One can upload custom patterns to the board to be injected into the trigger stream at a given point as well as reading out data as it passes through the chain.

rawDataUtils; This is a class that contains functions used to manipulate and crate pattern data files compatible with those injected at various points in the SP path. There are functions to read files in and write them out, read in LUTs and convert data that goes through them or reconstruct data that has been through them in reverse. Other patterns such as walking ones, walking zeroes can be selected and written to local files. The class contains a "main" function in which it is thought the user would implement their own system of calls and data manipulations.

registerSettings; This is a simple class to convert real world eta values to and from Track Finder eta cut values. It facilitates the construction of eta window keys used in Track Finder configuration.

3.3.3 Test Executables for SPFunctions

The test directory of the SPFunctions package contains short pieces of code that wrap functions found in the source code into useful executable functions. The functions can be executed as is but are deliberately written so as to make the process as transparent as possible to the user who may wish to modify them for his own means. They represent a rapid stand-alone method of accessing the hardware bypassing any run control or trigger supervisor implementation.

[3] At the time of writing (November 2008), the interface was defined but not all of the implementation existed for the calculation functions.

The executables in the *SPFunctions* package that start with *readCrate_* are all designed so that they can be run while the crate is actively running and triggering without interfering. They do not alter any settings and typically serve only to read out some stored information from the crate.

readCrate_FWnConfig.exe; This is an executable that will read out the firmware dates of all of the Track Finder chips as well as the configuration key date string of all of the SPs. The core objects are built in "debug" mode so the process is robust against incorrect firmware, lack of clock and bad configuration as far as possible.

readCrate_QPLL.exe; This executable returns the lock status for the SP QPLLs and the CCB. Typically if any of them are unlocked then there is a distribution problem with the TTC 40.08 MHz clock that will have to be resolved before much meaningful work can be done with the Track Finder Crate.

readCrate_AF.exe; This program returns information about the Alignment FIFO delay settings for each of the SP and the word count in the Alignment FIFO elastic buffer for each trigger link from the last resync. The output is a table where the columns represent Sector Processors and as one reads down one sees information about the board and then about each trigger link in order. There is a vector of output before the table that corresponds to the Alignment FIFO delay setting minus the word count divided by 2. This is a popular set of numbers to plot against fiber length. See [4] and [5] for more details on the Alignment FIFO elastic buffer and trigger link synchronization using the resync.

readCrate_rates.exe; This executable read out the rate of the incoming trigger primitives on each of the trigger links, the track rate of each SP and the overall crate trigger request rate. The time base with which the reads from the electronics are made and the data is updated defaults at every 5 seconds but this is configurable via an optional parsed argument. The output is in terms of a table with a time stamp. The executable can be run in a session window for a live read out of the current rate or piped to a file to form a rate history log.

readCrate_triggerLinkIDs.exe; An executable to read out the link IDs and muon number associated with each trigger link. The MPC carries a "peripheral crate ID" unique to each peripheral crate. This, along with the "Muon Link Number" (1,2 or 3) is transmitted over the optical fiber on each resync. This gives a unique identifier to each incoming trigger fiber that can be checked against the expected values and hence allow reporting of mis-mapped fibers. The output of this program is a map of the trigger fibers and a report of the incorrectly fibered links (as well as a suggestion as to how to fix the problem). This program should be run after any fibering work. There are more details on the fiber mapping in [5].

readCrate_ValPats.exe; This is a simple executable to read out the current values for the the valid pattern counter on each of the trigger links. This counter simply counts primitives as they come to the SP from the peripheral crates. The functionality here is much repeated in the more complex readCrate_rates which gives a rate rather than the simple instantaneous count.

readDAQFIFO.exe; This is a simple bit of code that reads out the data stored in the DAQ FIFO. One can can parse optional arguments of the SP slot number and desired output file name otherwise defaults are used. The output is an ascii file containing a dump of the raw data in hexadecimal format. A second file is written out containing any link errors. It is important to note that the DAQ FIFO receives data each time there is a trigger. Once it is full then it stops taking any new data. It is for this reason that it is often prudent to run the *readDAQFIFO* program once to clear the FIFO, let it fill and then run it again to take the data. If the file names are the same then the old one is simply overwritten. The raw data can be unpacked using the tools in the *rootUnpackSPDAQ* package described in 3.7. If one wants to continuously read data over VME then one uses the next executable; *takeVMEData.*

takeVMEData.exe; This executable is similar to the one above except that it will dump VME data continuously (or for a specified number of reads). One can parse optional arguments of SP slot number and output file name otherwise defaults are used. One can further configure the number of reads (or set "continuous" mode) and the pause between each read, which should be tuned to the trigger rate. Configuration of these last 2 is done by direct editing of the first few lines of the code in */test/*

readSPYFIFO.exe; This executable dumps out files containing all of the data that is loaded into the SP SPY FIFOs. Be aware that there are almost 40 FIFOs and so there are a corresponding number of files that are written out.

test_TFCrateContainer.exe; This executable is intended to demonstrate the various modes in which a *TFCrateContainer* can be instantiated (see Table 1). The code inside is an example of the most simple process needed to build the container.

test_sTTS_states.exe; This code is intended to demonstrate the sTTS based tests that can be applied to an SP using the sTTS_states class. One can parse the chosen SP by SP number (1-12) directly or set it interactively.

test_monitorables.exe; The code represents and example showing how to use an *SPObjectParser* to get hold of an *SPobject** and then use it to read out a subset of the "monitorable items" status and counter information from the corresponding Sector Processor board.

test_CCB_TTCRX.exe; This executable is an example of how to use the member functions in the CCBfunctions class to read and write the registers in the TTC-RX chip. Specifically reading back of the unique chip ID and the getting and setting of the coarse and fine delays is shown. The TTC-RX chip is mounted on the TTC-RQ daughter board attached to the front face of the CCB at the point where the TTC fiber connects to it. It is possible to configure delays for the incoming clock and fast commands using these registers. More details are in the CCB manual linked from [8] and the TTC-RQ manual in [20].

PRBS_test.exe; This executable will put all of the SPs in the Track Finder Crate into PRBS[4] mode. It will reset the error counters and then repeatedly read out an error rate and total count for each of the trigger links, appending a system time stamp to each read. The output should be piped to a file. The regularity with which one makes the reads is set to 1 minute by default but can be modified by parsing an integer value in seconds. Typically one would set the MPCs into PRBS mode first (MPC register CSR0[8]) and then run this executable, piping the output to a log file. If one intends to run this for long periods then it is worth running using unix *screen* or *nohup* commands so that the process runs uninterrupted on a local session.

injectSPYData.exe; This short piece of code wraps the *SPYFIFOFunctions::injectSPYdata* function. It allows you to upload files of data to each of the front FPGAs on an SP and then inject that data into the trigger path on reception of a specific BGo. The paths to the pattern files to be loaded and injected and the slot number of the SP are coded in the *injectSPYData.cpp* file. It is expected that the user would edit this file with their own specifics and then recompile. The patterns are in the standard form of 2 16-bit ascii frames per bunch crossing, the format of which is described by the *DAT_TF* register in the SP manual [8]. One can fill the files with up to 512 bunch crossings worth of data (1024 lines). This function can be used to validate the patterns as they pass through the SP Track Finding logic, from the SP to the GMT and from the SP to the DTTF making this one of the most powerful tools in the commissioning armory.

injectMPC2SP.exe; This code injects a user specified file of data from an MPC in slot 5 of the Track Finder Crate to a Sector Processor chosen by the user and then reads out the contents of the SP SPY FIFOs. The code is an example of an executable that wraps several of the SPFunctions classes. It uses an *SPObjectParser* from the base interactions library to construct the board objects needed. The user can either parse the SP slot number and the full path to the data file (in that order) or can type them in interactively.

init_MS.exe; This code contains snippets to run the functions exported by the Muon Sorter Test code which are described in the MS user guide liked from [8]. Several MS functions are wrapped, including simple initialization and self tests as well as more complex preocedures such as MS to GMT and SP to MS data injection routines. All are enabled by default along with sensible relevant settings but it is very much foreseen that the user will open up the *init_MS.cpp* file and configure it to their needs before recompiling.

init_CCB.exe; This is a stand alone executable to initialize the Track Finder Crate CCB following the instructions set out in 1.1 of the User Guide[8].

initTTCVi.exe; This is a standalone executable to initialize a TTCVi[5] board in slot 3.

findRegisterSettings.exe; This is a simple stand alone executable that wraps the *registerSettings* class. Through following the user interface, one inputs real world eta or eta window cut values and is returned values corresponding to settings applicable to the Sector Processor "DAT_ETA" eta scale.

initSPforRealData.exe, initSPTightWindows.exe, and *initSPforME11EtaOnly.exe*; These are executables that wrap stand alone SP configuration functions in the *initialConfigurations* class. The difference is typically the setting of the *DAT_ETA* cuts which are applied in line with the executable name. These executables come with the warning that they are fairly old and the user is expected to modify parameters such as SP slot number by hand.

initSPforMPCtest.exe; This is a simple wrapper to the *initialConfigurations* class and serves to put the SP in a mode whereby it is ready to receive MPC test patterns.

calc_PFD.exe; This executable wraps the *timings* class and allows calculation and manipulation of SP Pipeline FIFO delay settings.

---

[4] Pseudo-random bit stream mode. The MPCs send a stream of pre-defined ("pseudo-random") data over the trigger links to the SPs which then catch and validate it bit for bit, counting link errors as they go. It is a simple initial test of link stability. It is worth noting however that the test is actually a function of the optical transceivers and so no actual inter-FPGA transmission is taking place, for that one needs the pattern injection routines.

[5] The TTC-Vi was a predecessor to the TTC-Ci. It was replaced for Track Finder usage during 2006. The nice feature of the TTC-VI/VX combination as opposed to the modern CI/EX was that one could talk directly to the registers using code similar to that shown here. The modern modules require the TTC software as an interface.

main.cpp and corresponding *main.exe*; This is a piece of example code. It shows inclusion of most of the SPFunctions classes, the standard way to get hold of board objects and how to use the 2 together to manipulate the electronics. It is intended that the user would use customize this executable template to their own purposes.

shootSPL1As.exe; This function allows the user to artificially create SP tracks (and hence potentially L1 trigger requests), SP level-1 accepts (as if the GT had sent a trigger) or a combination of both. The user is queried for SP number, the number of requests to make and the type of requests. This is probably that fastest way to set the SPs into a mode where they generate triggers and it was heavily used in the commissioning of the trigger chain.

runRawDataUtils.exe; This executable simply calls *rawDataUtils::main()*. The r*awDataUtils* class is one that contains many varied functions that allow the manipulation and creating of raw data patterns. It was thought that in this exceptional case, the user would probably like to edit their project directly in a member function of the class, "main". One would edit the source code file, recompile the *SPFunctions* package and then run the functions from the test directory using this simple function wrapper.

parseCommandFile.exe; This executable is an example of how one can get hold of an *SPobject** and parse a whole file containing a command sequence to it. This is a very simple and effective way of manipulating the hardware by hand. One has a file of commands open, one edits it, parses it to the hardware using the command parser, edits the file, re parses it etc. Inside the code block of *parseCommandFile.cpp* there is a commented out section that if enabled would demonstrate a way of directly passing a sequence of commands from a C char string.

3.4 Lookup Tables and Firmware; SPLUTsAndFirmWare Package

3.4.1 Lookup Tables

This package deals with loading Lookup Tables (LUTs) to the SPs and firmware to the SPs, the Muon Sorter and the Clock and Control Board via VME[6].

Functions for SP lookup table manipulations are all contained within a single class; *loadLUTs*. Its header file is in *SPLUTsAndFirmware/include/loadLUTs.h*. This class provides functionality for loading LUTs to single chips as well as global wrappers for loading all the LUTs to a single SP or to the whole crate. It enables switching between binary and ascii mode for LUT file input and output and switches between VME block transfer and single writes mode. The class provides diagnostic routines whereby CRC checks can be run on loaded LUTs and the LUTs themselves can be dumped to local disk. The methods form this class are exposed through the dynamic library in the package and are picked up by executables in the test directory.

The most basic LUT loading function is *bool loadCrate(TFCrateContainer, pathToLUTFile)*. This function takes a Track Finder Crate Container object and then attempts to load the LUTs stored in the path pointed to by *pathToLUTFile* to all of the *SPobjects* instantiated in the *TFCrateContainer*. The function returns a bool which is set to true only if LUTs were loaded to all SPs and they all passed the CRC check (and the explicit verification of it is enabled).

The loadCrate function wraps a lower level function; *loadAllLUTs_SP(SPobject, LUTFilePaths_map)*. This function loads all of the LUTs to a single SP, a pointer to which is passed to the function along with a string-string map that contains local file paths to each of the LUTs of the type shown in the table below.

Table 2. A table showing the <string, string> map that must be filled with paths to LUTs in order to construct the second argument to be passed to the function to load all LUTs to a given SP (*loadAllLUTs_SP*).

Name of the LUT (std::string). The parameters in this column are fixed, you must not change them.	Path to that LUT (std::string). These you configure depending on where on the local disk the appropriate LUTs are found.
localPhiLUT_path	Path to the local phi LUT to be loaded to all 5 front FPGAs
DTME1ALUT_path	Path to the DT phi LUT to be loaded to front FPGA 1.
DTME1BLUT_path	Path to the DT phi LUT to be loaded to front FPGA 2.

[6] It is also possible to program the boards via Xilinx Parallel cable. See the relevant manuals in [8] and the commissioning note in [5].

Name of the LUT (std::string). The parameters in this column are fixed, you must not change them.	Path to that LUT (std::string). These you configure depending on where on the local disk the appropriate LUTs are found.
GPME1a_path	Path to the global phi LUT to be loaded to front FPGA 1.
GPME1b_path	Path to the global phi LUT to be loaded to front FPGA 2.
GPME2_path	Path to the global phi LUT to be loaded to front FPGA 3.
GPME3_path	Path to the global phi LUT to be loaded to front FPGA 4.
GPME4_path	Path to the global phi LUT to be loaded to front FPGA 5.
GEME1a_path	Path to the global eta LUT to be loaded to front FPGA 1.
GEME1b_path	Path to the global eta LUT to be loaded to front FPGA 2.
GEME2_path	Path to the global eta LUT to be loaded to front FPGA 3.
GEME3_path	Path to the global eta LUT to be loaded to front FPGA 4.
GEME4_path	Path to the global eta LUT to be loaded to front FPGA 5.
PT_path	Path to the Pt LUT to be loaded to the SP chip.

One can also generate a map similar to that above, filled with predicted values for the paths to each of the LUTs based upon the path to the LUT file folder and the SP number by running the function *<string, string> map workOutLUTFilePaths(pathToFolder, SPNumber)*.

The single SP load function wraps a set of lower level functions of the type *bool loadLocalPhi(SPobject, pathToLUTFile)* which serve to load a given LUT to the appropriate FPGA(s). These are also exposed publicly and can be employed directly by the user. Further, there is a set of very similar functions which have *_broadcast* appended to the name. These functions will simultaneously load the specified LUT to the corresponding FPGAs on all SPs via a broadcast to the backplane. This mode allows all SPs to have LUTs loaded in the time it would usually take to load one however one can only choose one set of LUTs.

There are several functions to do with CRC verification of loaded LUTs. The SP has the functionality to calculate a CRC from the currently loaded LUTs and compare the values returned to a set of values uploaded by the user. It is expected that for a standard LUT load, one would first upload the expected CRC results, then load the LUTs and ask the SP to make the comparison. Upon release of a new set of LUTs (or SP firmware), one would load the new LUTs, ask the SP to run the CRC check and then dump out the CRC monitoring data to be uploaded in future use. The specific functions in the loadLUTs class that deal with the CRC verification are shown in the table below.

Table 3. A table showing the CRC verification related function in *loadLUTs*.

Function	Notes
bool resetCRCMonitoringAndVerificationFIFOs(SPobject)	This function resets the counters and FIFOs associated with loading monitoring data and running CRC checks. It should be run prior to performing CRC related manipulations.
bool loadMonitorData(SPobject, vector<int> verificationData)	This function loads data corresponding to the expected CRC result. It is stored in the key each time a new configuration is created.
bool runCRCVerification(SPobject)	This is the function that actually makes the SP run the CRC calculation and perform the comparison of the results against the loaded expected data.

Function	Notes
vector<int> readBackVerificationData(SPobject)	This returns a vector corresponding to the verification data (result of the CRC check) that is in the SP.
vector<int> readBackMonitoringData(SPobject)	This returns a vector corresponding to the monitoring data (data loaded as expected result of the CRC check) that is in the SP.

More explicit verification of the loaded LUTs can be gained by setting the class switch *bool want_ExplicitLUTVerify* to true. If this is set then each time a LUT is loaded, all of the data loaded to all of the addresses is read back from the SP and explicitly checked against that stored in the file o be loaded. This is a very slow process and can magnify the LUT loading by a factor of around 5 and so the default is that this switch is set to false.

The switch *bool asciiFileMode* should be set to true when the LUT files are in ascii format and false when they are in binary. The class will switch automatically and throw a warning if it sees a file with an unexpected extension but on the whole it is best practice to set this variable straight after class instantiation time. The default is ascii mode.

In block transfer mode, the controller buffers up to 128 2-byte writes[7] in one go and then blasts them over the backplane in a single shot. It is much faster than single transfer mode but is potentially harder to debug. The class has a switch to turn block transfer mode on and off as *bool use_blockTransfer*. The default is true and so block transfer is indeed used for normal running.

As with most of the *csctf* packages, the *SPLUTsAndFirmWare/test* directory is one that holds executables compiled to include the base library and serves to hold examples of how to use some of the library functions. There are several example executables in this one;

crateLoadLUTs: This executable loads LUTs to the crate, runs a CRC check over them, performs explicit verification and then writes out the monitoring data that would be needed to create a key back in to the folder containing the LUTs. If one opens up the .cpp file then one can edit the path to the LUT set and see each of the above steps done through a member function call. The calls can be commented in or out as needed.

crateRunCRC: This code uploads a folder of CRC monitoring data (expected CRC results) to all of the SPs in the crate, gets them to run CRCs and then returns the results. The result will be positive if the LUTs already loaded in to the crate match those used originally to write out the monitoring data. The monitoring data is of the kind (ascii flat files, one per SP) written out by the crateLoadLUTs executable. The data inside the files is of the type produced by the stream from loadLUTs:: readBackMonitoringData(SP*) and in fact it is this function that is employed by the executable above in the dumping process. The user is expected to modify the first few lines of code to specify the location of the monitoring data folder.

dumpSPLUTs: This executable dumps out all 52 LUTs currently loaded in to the memory of a given SP. This executable takes a single argument of SP slot number.

compareLUTs: This executable compares a LUT set that has been dumped out of the hardware using the *dumpSTLUTs* executable above. The LUTs that are dumped out of the hardware are written in ascii mode by default and this is what is expected by this comparison function. The input in terms of the CMSSW dumped tables is binary since this is the CMSSW default. This executable takes 3 arguments; CMSSW input folder path, SP dumped LUT folder path and SP Number where SP Number is the standard 1-12 format.

replace_PTLUT_modex: This function takes a Pt-LUT in ascii (dec) format, searches for entries corresponding to a given track mode and then replaces them with a specified Pt LUT output entry. It takes 4 arguments; input Pt table file name(ascii,dec), output Pt table file name(ascii, dec), the mode to search and replace (hex), the value to replace them with (hex). An example execution would be;

./replace_PTLUT_modex.exe L1CSCPtLUT.dat L1CSCPtLUT_replaced.dat 0xf 0xffff

This would take '*L1CSCPtLUT.dat*' which is a PT LUT with decimal entries, finds mode 0xf (15) entries and replace them all with the value 0xffff (but in decimal i.e. 65535). The output file would be '*L1CSCPtLUTreplaced.dat*'. This code was used in early running when it was necessary to replace (nonsensical) singles and halo trigger type entries with a value that would signify valid Pt in the output to the MS and GMT and hence allow triggers on those modes.

[7] This limit is actually configurable in SPobject.h

test_loadLUTs: This is an example test executable designed to demonstrate several of the *loadLUTs* methods and the correct manner in which to employ them. It is intended to be extended by the user.

3.4.2 Firmware

For the Sector Processor, Clock and Control Board and Muon Sorter, code to upload firmware files to the boards exists in the *SPLUTsAndFirmWare/bin/LoadFPGAconsole*. For the DDU, one uses the HyperDAQ interface provided by the FED software to browse for and upload firmware files.

An SPobject created in standard mode (see Table 1) will check the date stamps of the loaded firmware in all chips on the Sector Processor. If there is a mismatch between the expected firmware date and the one read then an error will be thrown and the SP will not be created. If one uses debug mode then a warning will simply be thrown and the object creation will continue regardless. The reference for the expected firmware versions is either parsed as part of the configurations key read out from the configurations database or in the case of standalone mode is read from the local file *SPBaseInteractions/firmwareVersions.txt.*

Typically new firmware is uploaded on on the advice of the board engineers and experts who will provide a link to the new firmware file. There is a template directory in *SPLUTsAndFirmWare/firmware* in which firmware files can be stored before being uploaded (and some versions are indeed checked in to CVS in this directory) but in principle one can upload files from anywhere on the local disk. The main upload scripts are in *SPLUTsAndFirmWare/scripts/* where the script names denote the board to which they apply. Typically, one edits the loading script with the path to the correct firmware files and then runs the executable. Sufficient examples exist inside the scripts for the users to be easily able to add in new firmware but the form of the direct call to LoadFPGAConsole is given below nonetheless;

./LoadFPGAConsole -s8 -f1 -o1 -j1 -xAddressTable.xml my_chain1_fw.svf
./LoadFPGAConsole -s8 -f1 -o1 -j0 -xAddressTable.xml my_chain0_fw.svf

This code would load *"my_chain1_fw.svf"* to the chain 1 chips (Front FPGAs, VME FPGA, DDU FPGA) and *"my_chain0_fw.svf"* to the chain 0 (SP chip) to the SP in slot 8 using the registers defined in *"AddressTable.xml"*. If one looks inside the loading scripts then it is evident that it is essentially these 2 lines which need to be edited for the path to the most recent firmware file.

The method of upload using the VME interface (via the control PC and nfs mounted account software) is using JTAG Boundary Scan. Firmware files to be loaded to the SPs are Serial Vector Format (.svf) and are an open format ascii representation of JTAG test patterns. The SPs take 2 separate .svf files; chain-zero is the firmware loaded to the SP core chip and chain-1 contains the commands loaded to the rest of the chips. The uncompressed file sizes are typically of order 8.5 and 14MB respectively but vary a little from release to release. Code to load firmware to the Muon Sorter also exists and in principle the CCB can be programmed using the same boundary scan method. Each of these boards has only a single chain.

It is also possible to load firmware (.mcs files) via a Xilinx Parallel Cable IV directly linked to a Windows laptop running the (free) Xilinx software package "Impact" [19] and corresponding 14 pin connectors on the boards. Further detail of loading MS and CCB firmware is given in the user guide linked from [8].

There are further details on both LUTs and firmware in Section 6 in the note referenced from [5].

3.5 SPValidation

The SPValidation library represents a suite of code to debug Sector Processor hardware and firmware. It is intended that each time a new module is produced, an old one is repaired or a new firmware set is uploaded that the basic tests would be run before any higher level processes incorporating the module are executed. For most of the executables in SPValidation, one can obtain a listing of the arguments and functionality by parsing a *-h* argument.

3.5.1 TestSPConsole

The main test executable is *SPValidaion/bin/TestSPConsole.* Typically one runs it, parsing only the slot number corresponding to the SP under test. The user is faced with a menu interface as shown in Figure 4, below.

```
[csctfts@vmepcS2G18-10 SPValidation]$ bin/TestSPconsole -s21
SPObjectParser --> debug mode; errors will be inhibited..
CCB04 f/w date ==05/03/07

Available Validating Procedures for SP2002:
   1) Hard Reset Test
   2) IDTB Test
   3) Clock Test
   4) Chip ID Test
   5) Link Loopback Test
   6) Lookup Table (LUT) Test
   7) Readout Test
   8) Test Front FPGA lines to SP

   9) Perform All Tests

   0) Quit Program
```

Figure 4. The TestSPConsole main menu.

Each of the tests themselves broadly maps to a class in the SPValidation source code. They are described in Table 4, below. Typically the user would simply select option 9 in order to run a Sector Processor through all of the validations.

Table 4. The contents of *SPValidations/bin/TestSPConsole.exe.*

Test	Description
1/ Hard Reset Test	The SP performs a hard reset, firmware is reloaded to the FPGAs from the EEPROMs and the status of each of the chips is pulsed afterwards. The test is passed if all of the chips come back successfully. During the test, one can see the little LEDs on each of the chips go out and then come back on. If all is well then the chips show green LEDs, if not then red ones come on.
2/ IDTB Test	This test validates the Internal Data Transfer Bus (IDTB). It is the synchronous parallel transfer bus that is responsible for communication between the VME FPGA and the other FPGAs. This test uploads patterns of walking 1s and 0s and then tests their transfer between FPGAs. If the patterns are transferred without error then the test is passed.
3/ Clock Test	This test reads out the status of the various clock related registers. There are 2 digital clock managers relating to the 2 SP clocking domains and lock status registers corresponding to the QPLL daughter board. If any of these are in a state not compatible with standard running then the test is failed.
4/ Chip ID Test	The chip ID test really refers to the simple read out of the firmware dates of the SP FPGAs. It is a single register read for each one and is a simple test of a handshake with a control & status register.
5/ Link Loopback Test	The loopback test mode actually offers 2 options; the first to perform a loopback using the TLK opto-transceiver only and the second to use an optical fiber to loop the data back. For both cases, patterns are loaded to the user selected front FPGA or DD chips and then injected in to the trigger chain to be read out by the spy FIFOs at the VME and SP chips and validated. For the simple *"Tlk Loopback"*, the path is truly internal to the SP, for the case of using a fiber then the patterns are actually transmitted and received by the SP before being passed through to the central chip FIFOs.

Test	Description
6/ Lookup Table Test	This test loads and reads back a set of dummy LUTs of the kind specified by the user. It validates all the address and data bus lines for the LUTs. The test is passed if the pattern read back matches the sample pattern loaded.
7/ Readout Test	This test loads and injects patterns from the front or SP FPGAs to the DD chips data recording FIFOs. The user chooses the chip to be the progenitor and the test is passed if all of the patterns are transferred successfully.
8/ Test Front FPGA lines to SP	This test examines the lines from the front FPGAs to the main SP FPGA. A series of pattern injections are made at the board front end and are read out from the SP spy FIFOs.

3.5.2 DTLoopBackTest

The executable *SPValidation/bin/DTLoopBackTest.exe* provides a suite of tests for validating the Drift Tube Transition Board modules. The test essentially involves loading a set of patterns to the SP front FPGAs, injecting them into the transition board and then reading back out the pattern. One has the option of running a purely internal test or actually using a SCSI cable to link inputs and outputs of the transition board and then include them in the injection path. The method for connecting the loopback cables and external transfer board is described in Appendix 2 of [5]. The user has to tune the loopback latency in terms of the number of clock cycles the loopback path takes. Options to inject walking 1s, 0s, random or user defined data exist as well as options to repeat the test continuously allowing stress testing of the links. Automatic detection of old pre-production DT transition board prototypes and associated register switching exists within the code. An example line used to run the executable is shown below.

./DTLoopBackTest -s6 -c100 -mr -l5 -do -p1

This line would load random patterns (*-mr*) to front FPGA1, "path 1" (*-p1*) on the SP in slot 6 (*-s6*). They would be injected via the external loopback (there is no *-i*, "internal" argument) and the latency of the loopback path is 5 clock cycles (*-l5*). The log is written to the screen (*-do*) and the test is repeated 100 times (*-c100*).

3.5.3 MPCSPtest

This is a simple executable to load patterns to the MPC and then inject them to the SP where they are read out. It was used as part of the early commissioning of the prototype SPs in 2006. As such it would require some modification by the user to be used on the final stands. Much of the code is replicated in the SPFunctions library but this is never the less a simple well encapsulated test executable and so it still resides in CVS as an option.

3.5.4 SPDDUcomrehensive

This executable automates the loading of test patterns in to the SP and then manages their injection via the optical links to the DDU. It probes for possible errors in the DAQ readout path of the Track-Finder crate. It injects simple test patterns to SPs, reads back resulting event records from the DDU and then verifies the structure and content of these events. The injection rate can be adjusted by cursor keys while the program is running.

This executable is reasonably old (2006) but the automated readout of the DDU is not replicated in the rest of the Track Finder software and so it is left in this package. The DDU typically now comes under the control of the FED Software package.

3.5.5 dumpConf

As the name suggests, this executable serves to dump out pertinent configuration register states for the SP parsed.

3.5.6 SPtester

This is a program to test the entire Sector Processor board logical chain. It injects test patterns (random or walking 1s and 0s) to the Front FPGAs on the board, reads back hardware output and then compares it to the SP offline simulation. It was used in the commissioning new hardware versions of Mezzanine cards.

3.6 Stand Alone Executables

The *csctf/SPStandAlones* area is where code to directly read and write Track Finder Crate registers resides along with short scripts wrapping the calls into functional sets. The main set of functions reside in SPStandAlones/test/ readWriteRegisterFuncts. These executables are stand alone, there is no separate source code in this package. These read-write-register functions allow the user to directly read from or write to registers one by one and as such are assumed to be expert domain only. There are functions for reading and writing registers in the Sector Processor, the Clock and Control Board, the Muon Sorter and the TTC-Vi. The functions make use of SPobject* instantiated in "debug" mode which means that errors are suppressed as far as possible and are only presented as warnings. The piece of code below shows an example write to an SP.

./readwriteRegister -s7 -rCSR_PFD -cSP -mMA -pW -v0x5

The call above would write (*-w*) 0x5 (*-v0x5*) to the CSR_PFD register (*-rCSR_PFD)* on the SP chip (*-cSP*) of the SP in slot 7 (*-s7*). The corresponding way to read back the same register would be;

./readwriteRegister -s7 -rCSR_PFD -cSP -mMA -pR

A full listing of the arguments that one can parse can be obtained by running the executable without arguments. Variables such as the bus adapter type and xml register base offset table path are hard coded and so one would need to directly edit the code and recompile to change them.

Within the *SPStandAlones* directory, there are 2 more folders; *SPStandAlones/SPScripts* and *SPStandAlones/ TTCviScripts*. These hold simple scripts that wrap the readwrite register calls to produce small functions in the SP and TTC-VI boards respectively. They are typically old and so any new user could use them as a guiding basis for reproducing the function but would be well advised to check each register call one by one.

3.7 Local Unpacking of Data without using CMSSW and a few other useful tools; rootUnpackSPDAQ

The *$CSCTF/rootUnpackSPDAQ* folder contains a set of GCC compiled executables that are intended as a few miscellaneous useful tools. Included are converters to switch between ascii and binary files and tools to unpack raw Track Finder data, perform a quick analysis and spit out some ROOT[6] plots. They are not dependent on the usual *$CSCTF* core libraries and as such can be much more easily compiled on various platforms and architectures. The only dependencies come from the data unpacking tools which do need ROOT[6]. If you want to use them then you need ROOT installed and working and the root *libCore.so* in the *$LD_LIBRARY_PATH* environmental variable list. In order to compile the executables, simply type *./compile.sh* to run the compile script.

myBin2Ascii and myAscii2Bin;

Two simple converters exist for conversion of files from ascii to binary and vice versa. They are *myAscii2Bin* and *myBin2Ascii*. Table 5 below shows an example where an ascii LUT in decimal is converted into binary and then back in to ascii but in hex format.

Table 5. An example of the use of the ascii to binary and binary to ascii conversion tools. One can switch *hex* and *dec* in the examples as the input or the desired output changes.

commands	notes
./myAscii2Bin.exe LocalPhiLUT.dat LocalPhiLUT.bin dec	Convert the (decimal format) ascii file *LocalPhiLUT.dat* in to binary file *LocalPhiLUT.bin*
./myBin2Ascii.exe LocalPhiLUT.bin hex > LocalPhiLUT_hex.dat	Convert the binary file *LocalPhiLUT.bin* to hex format and pipe it to the ascii file *LocalPhiLUT_hex.dat*

In the course of unpacking raw data by hand, the first step is to transfer the raw binary file in to ascii. If one is intending to use the stand alone unpacking tools in the rootUnpackSPDAQ directory then the data file should be unpacked into a hex format. The code would be of the form;

./myBin2Ascii.exe rawDataFile.bin hex > rawDataFile.dat

stripOutDDUEvent: The next step in the unpacking of data by hand process is to strip out on the data inside the raw file that comes from the SPs and ignore that from the DDU. The process *stripOutDDUEvent* performs just this task. One simply passes the arguments for (1) the input and (2) the output files with an optional (3) of start event number if it is not zero. The code form would be similar to that given below;

./stripOutDDUEvent.exe rawDataFile.dat rawDataFile_SPonly.dat

makeTree: Once one has an ascii raw data file in hex format containing only SP data then one can use this local unpacker. One passes a single argument of the path to the raw data file and then the unpacker spits a translation of the unpacked bits line by line to the standard out as well as writing a root tree with some of the unpacked data in to the */temp* directory. If one really wants to do debugging of the raw data line by line by eye then one should pipe the standard output to a file otherwise it is best to pipe the output to */dev/null* for speed. It is further worth noting that this file contains not only the input raw data but also its unpacked description and hence will be several times larger than the original. One can edit lines in makeTree.cpp to limit the number of events over which the program should run. Typically anything over 100000 events with full 7 bx DAQ FIFO settings becomes unmanageable in terms of size both standard output and the written root tree[8]. The code below shows an example of how one might run the code piping the output to a file;

./makeTree rawDataFile_SPonly.dat > unpackedData.txt

An example of the standard output is show in Table 6 below.

Table 6. An example event of the output of the raw unpacked data. The lines with '-----> 0xabc' are the lines of original data, the text lines under each block then translate selected information.

std out of makeTree for one event	notes
----------->> read word ::0x9207 ----------->> read word ::0x9025 ----------->> read word ::0x9000 ----------->> read word ::0x9601 ----------->> read word ::0xa000 ----------->> read word ::0xab54 ----------->> read word ::0xa008 ----------->> read word ::0xa7ff ---> EventHeader L1ANumber == 0x25207 L1A_BXN == 0x601	The SP Event Header. It is characterised by 4 '0x9...' words followed by 4 '0xa...' words. The 2 parameters unpacked are 'L1ANumber' which is simply a sequential counter counting triggers and the 'L1A_BXN' which is the bunch crossing number upon which the trigger arrived. n.b. A machine 'orbit' corresponds to the time it takes a given bunch to travel all the way round the LHC, it is about 89 micro seconds. Within an orbit there are 3564 25nS bunch crossings which is what the bunch crossing number is counting here.
----------->> read word ::0x40ff ----------->> read word ::0x5 ----------->> read word ::0x42de ----------->> read word ::0x51a2 ---> Block Counters Track Counter == 0x2c0ff Orbit Counter == 0x28d142de	This is a set of counters relating to this event. The Track Counter is an internal SP counter that simply monitors the number of tracks formed which the orbit counter is a global time counter counting the number of orbits as described in the row above. When one combines orbit and bunch crossing numbers, one has a unique reference to a given bunch crossing within a lumi segment.

[8] This is a tool better suited to by-hand debugging of raw data than processing of large numbers of events. If however a large run is required then one can simply run several processes in series and if memory permits chain the root trees together on the fly in subsequent analysis.

std out of makeTree for one event	notes
...3 empty data block headers stripped out for brevity.	The DAQ FIFO is set to a 7 bx wide window. The SP Pipeline FIFO delays are tuned so that a self trigger arrives in the central slot. In this run there were no external triggers and so the first 3 bunch crossing slots were empty. I have removed the corresponding empty data block headers.
----------->> read word ::0x200 ----------->> read word ::0xb ----------->> read word ::0x0 ----------->> read word ::0x0 ----------->> read word ::0x0 ----------->> read word ::0x200 ----------->> read word ::0x1ff ----------->> read word ::0xff ---> DataBlockHeader valid pattern word == 0x200 Valid Quality MB-D == 0x0 Valid Quality MB-A == 0x0 Mode-1 == 0xb Mode-2 == 0x0 Mode-3 == 0x0 Synch Error word == 0x0	This is the triggered bunch crossing data block header, it tells you about the LCTs that came in on this bunch crossing and any output tracks that were sent to the MS. Here the 'valid pattern word' shows 0x200, this corresponds to a ME LCT coming in on the 'muon-1' trigger link of ME3. Had there been stubs from the DT Track Finder used as part of a track then they would have shown up in the 'Valid quality MB' lines. One sees a value of 0xb in the mode 1 signifying that a single track was found (first of 3 potential outputs of the SP). The value 0xb signifies that it was a track of type 'track mode 11' which corresponds to a singles trigger. Appendix 2 gives the full table of modes assigned to various track types.
----------->> read word ::0x12b3 ----------->> read word ::0x3861 ----------->> read word ::0x255c ----------->> read word ::0x20d5 ---> MEDataRecord, bx=3 LINK=a CLCT_PAT == 0x3 quality == 0xb wireGP == 0x12 CLCT_PAT_ID == 0x61 cscID == 0x8 L_R_bit == 0x1 BC0 == 0x0 BXN0 == 0x1 ME bx number == 0x55c	The ME-data record describes the incoming LCTs listed in the data block header above. Here we see the description of the data contained in the single ME3 LCT listed above. The CLCT pattern type and the quality of the LCT are parameters assigned by the TMB. The wire group number, strip pattern ID and CSC (chamber) ID are functions of the chamber and the location of the hit within it and are assigned by the on chamber electronics. The ME bunch crossing number is assigned by the TMB and in principal should be tuned so that the SP assigned bunch crossing number (header word, top row) matches that assigned by the TMB.

std out of makeTree for one event	notes
----------->> read word ::0xb18 ----------->> read word ::0x2000 ----------->> read word ::0x20 ----------->> read word ::0x0 ---> SPDataRecord, bx=3 mode=1 Phi == 0x18 sign == 0x0 CHRG == 0x0 ETA == 0xb Halo == 0x0 del12Phi == 0x0 del23Phi == 0x0 MS_ID == 0x0 MB_ID == 0x0 ME4_ID == 0x0 ME3_ID == 0x1 ME2_ID == 0x0 ME1_ID == 0x0 MB_TBIN == 0x0 ME4_TBIN == 0x0 ME3_TBIN == 0x0 ME2_TBIN == 0x0 ME1_TBIN == 0x0	This is the SP Data Record. It gives information about the track found by the SP. In this case the track was a singles track requiring one LCT only and so values such as the inter-station phi bend 'del12Phi' and 'del23Phi' are not filled. There is a single LCT in the track, it is the ME3 LCT listed above and values for eta and phi are filled since even though in this case there is no actual inter-station extrapolation, their values can be drawn from the LCT itself directly. The TBIN fields should be ones used to signify the timing of the incoming LCTs that were used to build the track however this functionality was not implemented as of November 2008.
...3 empty Data Block Headers excluded for brevity.	As with the 3 empty Data Block Headers in the 7 bx window before the triggered bunch crossing, the 3 after it were also blank.
----------->> read word ::0xf007 ----------->> read word ::0xf07f ----------->> read word ::0xf088 ----------->> read word ::0xf630 ----------->> read word ::0xe00c ----------->> read word ::0xeb54 ----------->> read word ::0xe823 ----------->> read word ::0xe971 ---> SPEventTrailer	This is the event trailer, signifying the end of this event for this SP. It is characterized by 4 '0xf...' words followed by 4 '0xe...' words.

The actual translation of the raw data is given in [7]. It is important that the *dictionary* classes holding the data translation are kept up to date with data format changes coming with SP firmware upgrades if the functionality of the local unpacker is to be preserved.

Once the program has run, on top of the std output of the kind shown in the table above, there is also a root file containing a single root tree in */temp/SPDAQTree.root.* One can easily browse its contents using a *TBrowser* within ROOT. In order to add to what is filled in to the tree, one would need to edit the *rootTree* class files so that new variables and corresponding tree branches are added and written out each event.

fastAnal_compiled: This is an executable meant as an example to show one loads the root tree written out by the raw data, loops over the events inside it, makes quantitative cuts on the variables inside it and writes out a root tree full of filled results histograms. The user can either use this code as a template to write their own root based analysis or tweak the code inside slightly and recompile as necessary. In order to run the default sample code, one simply types *./ fastAnal_compiled* (no arguments). The code searches for the *.root* file in the temp directory, loads it and then runs over parts of it as specified in the user interface. Root histograms are written into an output file also stored in the */temp* directory.

The final tool in the analysis chain is a script to automatically plot the histograms saved in the root file form the analysis. For the plots produced by the *fastAnal_compiled.exe* executable, the script is *show_fastAnalPlots*. This script

simply reads in the histograms saved in the file and builds plots on the basis of user configurations that live inside the script.

There are 2 other analysis routines that are based around the same principles as the one above. The executable timePlots.exe combined with the root script *show_timePlots* spits out some plots based around stub timing within the trigger data. The second is a *fastAnal.C* which when combined with the *startMe_fastAnal* script will run a quick analysis in the Root interpreter CINT. This is an extremely slow way to do analysis and so absolutely not recommended unless one is really debugging single events by hand using ROOT. The example is nevertheless left for the user.

3.8 TFHyperDAQ

The *TFHyperDAQ* or "Track Finder HyperDAQ" package is a legacy interface left over from the MTCCII (2006) commissioning period. It uses the XDAQ framework to provide a web page which wraps key functions exported from the CSCTF base libraries. It is essentially a way of gathering together the most important standalone executables in a simple user interface. More information on this interface and details for its use are in the "CSC TrackFinder shift Manual", July 2006, which is linked from the group page in [21].

4 The Trigger Supervisor Package[15]

The trigger supervisor part of the CSCTF online framework resides in *TriDAS/trigger/csctf/ts*. This forms the "CSC Track Finder Trigger Supervisor Cell" and is responsible for the connection between the Track Finder control software (described in this note) and higher level CMS level-1 trigger control. The code inside */TriDAS/trigger/csctf/ts* controls the hardware through functionality exported by the packages described in Section 3 and implements a SOAP[16] interface with the "Trigger Supervisor Central Cell" located in *TriDAS/trigger/ts.*

4.1 MTCCIIConfiguration[9]; Configuration using the Trigger Supervisor

The function MTCCIIConfiguration::f_configure() is the one that is called by the Trigger Supervisor package when Trigger Control wants to configure the hardware. The class holds a parameter list[10] that relates key attributes of the configuration process to their chosen settings. This list essentially forms the set of variables that are inputted to the process that configures the hardware. They can be configured via the SOAP command that is parsed from the Trigger Supervisor Central Cell or interactively via the local Cell HyperDAQ interface.

Table 7. The list of configurable parameters read in by the configuration process.

Parameter	Notes
KEY	This is the primary CSCTF configuration key parsed from the Trigger Menu. It is used to decode the secondary keys for each of the SPs using the primary configuration database. The secondary keys are then used on an SP by SP basis to query the secondary configuration database for the appropriate settings which are then parsed to the hardware. This is often the only parameter to be actively parsed, allowing default values to define the others.
Force Configure	This is a boolean choice which defines whether the hardware must be actively configured or not. If it is set to false then if all of the SPs already return the correct configuration key then no further configuration will be performed. If it is true then each board is actively configured from scratch regardless of its current state. The suggested default for this choice is false in order to reduce unneeded configuration process.
want_MultiThread	If this boolean choice is set to true then the main CPU thread will fork off 12 child threads, one for each of the SPs. Each thread is responsible for the configuration of a single board. The main thread registers and waits upon the exit of the child processes. This choice speeds up full crate configuration by a factor of about 3. It is recommended that the default is set to true.
want_HardCoreLUTAddressCheck	This choice allows the process that loads the LUTs to check all of the data values loaded into SP memory after each LUT configuration. It slows down the configuration process significantly but acts as a 100% backup to the LUT CRC process. The suggested default for this mode is false, it is really a debug utility.
want_LUT_CRC	This boolean switches on and off the ability of the SP to calculate a CRC on the recently loaded LUTs. The process takes about 10-12 seconds per SP to validate all of the LUTs by CRC and return a response. The recommended default setting is to enable the CRC.

[9] The name is misleading, this really is the class that is responsible for overall configuration of the hardware. The nomenclature originates from the MTCC commissioning period in 2006 when the Trigger Supervisor external configuration functionality was first developed and the CMS trigger group have never changed it.

[10] The parameter list is a little bit like the XDAQ version of a std::map<string, Type T>. Here rather than use std::string, the key and data field is actually xdata:String and the data field is one of xdata: Boolean, Integer or String.

Parameter	Notes
want_writeOut_KEYContents	This setting enables local writing of the contents of the keys read from the database as the configuration progresses. It is a useful debug tool but greatly adds to the size of the log file and so the default is false.
want_confiure_CCBnMS	The CCB and the MS do not have entries in the configuration key since their configuration is invariant and is defined in the users guide.[8]. The default option is that the MS and CCB should indeed be configured as part of the configuration of the Track Finder Crate and in fact form the first few steps of the process.
max_loadAttempts_perSP	This integer defines the number of attempts the process makes to load an SP that is failing in some way. During commissioning, the default value was set to 5 attempts but one could envisage that this will be reduced to 1 once the system is in a stable state.

The configuration chain is shown in Figure 3. Trigger control chooses a specific key with which to configure the CSC Track Finder. This key is parsed to the local cell via within the configuration SOAP message (or the default key is used). The local cell receives this key and then queries[11] the primary configuration database. The primary configuration database simply consists of a table that holds secondary configuration keys, one for each of the SPs. The secondary configuration database consists of tables containing data corresponding to SP configuration settings. These data are read in as each SP is configured and are used to parse settings to the SP and then to verify it.

The *TrackFinderCrateContainer* object that is responsible for holding the *SPobject*s* used to read and write to and from the hardware is contained in the *CellContext* class and follows the principles of instantiation on demand. The *MTCCIIConfiguration* class holds pointers to many of the other csctf base functions objects which is what it uses to perform the hardware manipulations. The boards themselves have a register that is devoted to storage of the currently loaded configuration key. If, after all of the optimization steps, it is necessary to write to a SP then the first write is one to overwrite the already loaded configuration key stored in the hardware with dummy value 0xffff. A valid key is then only written back into the hardware upon successful completion and verification of the configuration process.

The steps involved in configuring the Track Finder Crate as part of MTCCIIConfiguration are shown in Figure 5, below.

[11] At configuration time, interaction with the database is done via a wrapper to the ORACLE occi.h interface. Setting up the keys in the database in the first place is done via an SQLPlus script.

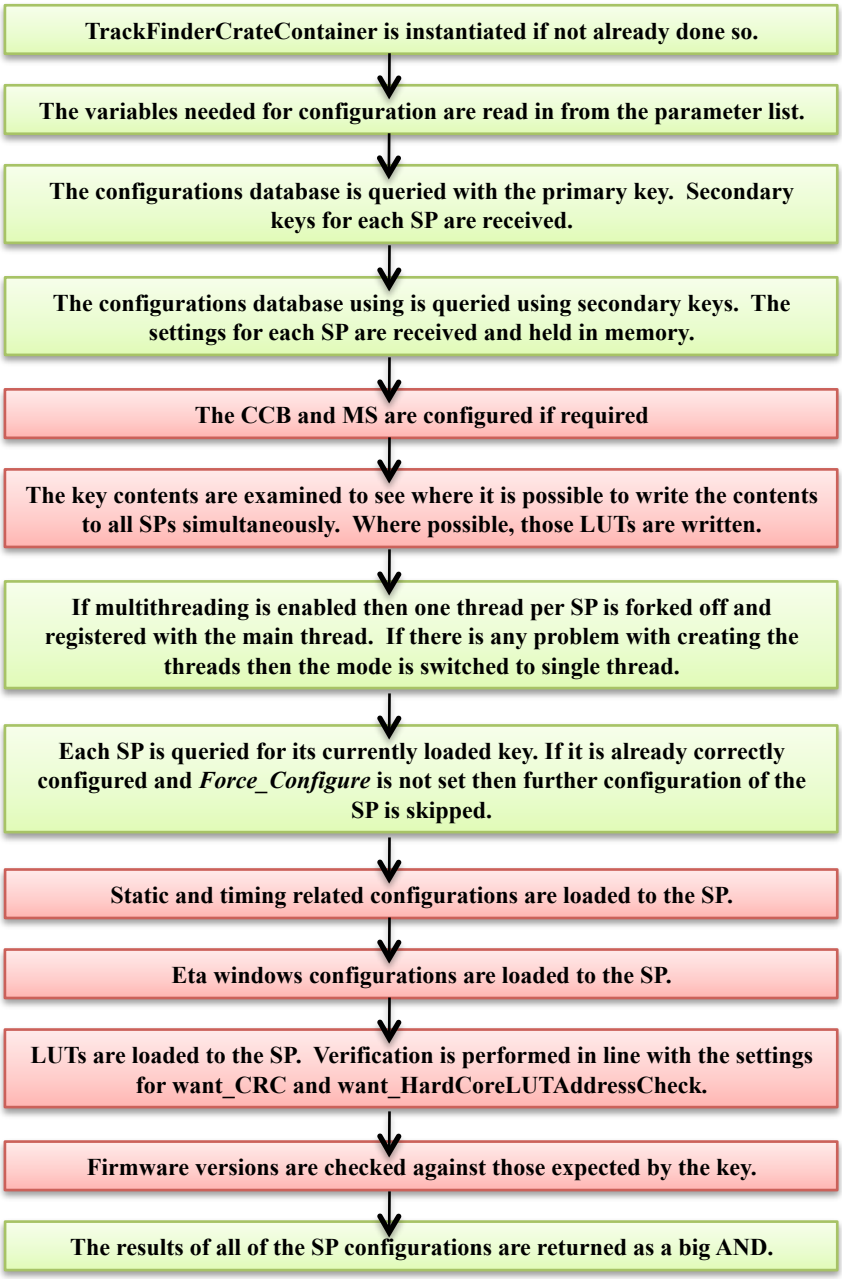

Figure 5. The major steps of the configuration process shown in sequence.

As the configuration process proceeds, the log output, along with appropriate time stamps and timings are written to a local configurations log.

There is further discussion on the content of the configuration database, the configuration process and an example key used for LHC start up in [5].

4.2 Test Functions Available through the Trigger Supervisor Interface

It is possible to put the Track Finder Crate in almost any desired test mode via the configure function described in 4.1. One simply has to set up the key set so that it contains the register settings appropriate for the test and then use the configure function to parse the contents of the database to the hardware. For a few specific tests, however, standard functionality from the SPFunctions library has been exported to the Trigger Supervisor interface. These provide the trigger community with a standard set of simple diagnostic tools which can be run to diagnose the L1 trigger without

any real input from specific hardware experts. For the CSC Track Finder, these tests do not use information stored in the database although they do often take xdata arguments.

The functionality of the *SPFunctions/sTTS* class is exported to the Trigger Supervisor Cell. The user can read or write a given sTTS state to a given FED-ID within the Track Finder Crate. One can also use the scrolling test functions or diagnostic global dumps.

The test injection mode from *SPFunctions/msConfigs* where the MS is configured to inject patterns to the GMT is exported. One can define the set of patterns to be injected or use the defaults and hence once has a rapid automated test of the downstream part of the Track Finder trigger path.

4.3 "Monitorable" Items and Online Data Quality Monitoring

The Trigger Supervisor and XDAQ packages provide a framework in which the hardware state can be monitored in real time. The class *TriDAS/trigger/csctf/ts/CSCTFDataSource* provides an interface through which the XDAQ framework can generate periodic requests to the local cell which then uses the CSCTF base libraries to query the hardware for a set of counters and status registers. The information is recorded in a "flash list" which is in essence a centralized store that can be queried in real time by multiple users and is also sent on to the conditions database where it is recorded for incorporation into offline analysis. The *CSCTFDataSource* class contains the configuration for the periodicity of the pulses (currently every 40 s) and the list of items to be read out from the electronics and fed on to the flash list and database.

One of the principal users is the L1 Trigger Monitoring application. This service provides real time graphical representation of the status and activity in the L1 trigger. On top of this, the CSC subdetector has a custom rack of machines in SCX[4] that are used for online analysis of CSC local data. This includes a machine that processes trigger data and automatically publishes the results to the web. Within the CSC community, these 2 services are colloquially known as "global" and "local" DQM respectively.

Further information on the CSCTF Trigger Supervisor Cell and details on using the local interface can be found in a "CSCTF Quick Start Guide" which is linked from the general CSCTF resources page[21]. From the same resource on can also find links to L1 trigger and CSC DQM web based services.

5 Exceptions

Exception handling in the csctf online framework is based around the XDAQ *xcept/Exception* class. The class used to raise exceptions is *SPBaseInteractions/include/TFException* and it simply inherits from the XDAQ version. This handler provides a standardized way to pass a description of the problem, the module, class and code line producing the error as well as a wrapper for the C++ <*exception*> or <*stdexcept*> standard exception references. TFException is placed in the SPBaseInteractions package since code for the core objects in here (as well as in the functions libraries) depends on it and it is necessary to avoid circular dependencies in shared library compilation.

The code at the core of the SPBaseInteractions library that is responsible for the interaction with the hardware drivers is further proofed to handle specific hardware access type exceptions. The most common are specific hardware diagnostic errors thrown by the XDAQ HAL package or the CAEN device driver. Typically the low level error is caught, wrapped with a more descriptive message and then rethrown as a XDAQ exception. There are no catch-all (*catch(...)*) in the low level code, the emphasis being with relying on the framework to handle the problems.

Higher level code such as the */test* standalone executables does try to replicate some of the exception handling functionality of the high level framework and so there are more generic catch statements and exit functions. These never overlap with code that could be implemented within the framework itself.

6 References

[1] **CMS Internal Note 1999/060.** Acosta, Madorsky, Scurlock, Wang, Atamanchuk, Golovtsov, Razmyslovich. *"The Track-Finding Processor for the Level-1 Trigger of the CMS Endcap Muon System".*

[2] **Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. Volume 496, Issue 1, 1 January 2003, Pages 64-82.**
Acosta, Adams, Atamanchouk, Cousins, Ferguson, Golovtsov, Hauser, Madorsky, Matveev, Mumford, Nussbaum, Padley, Razmyslovich, Sedovd, Smith, Tannenbaum. *"Development and test of a prototype regional track-finder for the Level-1 trigger of the cathode strip chamber muon system of CMS".*

[3] **CERN/LHCC 2000-38. CMS TDR 6.1 15 December 2000.** The CMS Collaboration. *"The TriDAS Project. Technical Design Report, Volume 1: The Trigger Systems".*

[4] **CMS Internal Note 2008/045**. Acosta, Gartner, Holmes, Jackson, Kotov, Madorsky, Uvarov, Wang, Geurts, Gilmore, Matveev, Rakness. *" The CSC Track Finder Installation".*

[5] **CMS Internal Note 2008/53**. Acosta, Gartner, Holmes, Jackson, Kotov, Madorsky, Uvarov, Wang, Geurts, Gilmore, Matveev, Rakness. *" Commissioning of the CSC Track Finder".*

[6] **ROOT; An object oriented data analysis framework**. Brun & Rademakers.
http://root.cern.ch/

[7] *"SP-to-DDU Event Record Structure".* Uvarov, University of Florida.
http://www.phys.ufl.edu/~uvarov/SP05/SP05.htm

[8] **Web links to detailed specification of the Track Finder Crate boards;**

Most of these boards have non-published manuals available from various HEP group web pages. Current web links are provided below. The links themselves are liable to change, the URL stems less so.

Sector Processor, DDU Extender Board and DT Transition Board. Built by the University of Florida CMS HEP group.
"SP Backplane Interfaces". Petersburg Nuclear Physics Institute / University of Florida. July 1, 2006, Version 8.3.
"CSC-DT Data Exchanging Transition Board Upgrade." L. Uvarov, V. Golovtsov. May 2004.
http://www.phys.ufl.edu/~uvarov/SP05/SP05.htm

Device Dependent Unit. Built by Ohio State University CMS HEP Group.
http://www.physics.ohio-state.edu/~cms/ddu/index.html

Clock and Control Board and Muon Sorter Board. Built by the CMS HEP Group at Rice University, Texas.
"Clock and Control Board for the CSC EMU Peripheral and Track Finder Electronics." CCB2004 Specification (Production Board). Rice University, Version 3.4, 27th June 2007.
"The MS2005 Muon Sorter Specification." Rice University, Version 1.3, 26th June 2007.
"The CCB2004, MPC2004 and MS2005 User's Guide". Rice University. V1.4.4 October 1st 2008.
http://bonner-ntserver.rice.edu/cms/projects.html
http://bonner-ntserver.rice.edu/cms/boards.html
http://lhc-workshop-2005.web.cern.ch/lhc%2Dworkshop%2D2005/Posters/80-MikeMatveev.pdf

[9] **Concurrent Versions System (CVS).**
http://www.nongnu.org/cvs/
Link to the CERN CMS TriDAS root;
http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/TriDAS/?cvsroot=tridas

[10] **DOXYGEN; Source code documentation generator tool.**
http://www.doxygen.org
A link to the DOXYGEN output of the Track Finder online software;
http://cern.ch/dan.holmes/documents/doxygen/index.html

[11] **XDAQ; CMS-CR 2003/007**, Gutleber et al. *"Using XDAQ in Application Scenarios of the CMS Experiment".*

[12] **CAEN 2718 VME Controller.**
"Mod. V2718 VME PCI Optical Link Bridge. Technical Information Manual, Revision 4". CAEN S.p.A. February 2005.
http://www.caen.it

[13] **SBS Technologies.**
http://www.sbs.org
SBS virtual bus adapter;
http://www.gefanucembedded.com/products/family/28

[14] **CERN Scientific Linux (SLC).** Open source Linux distribution based on Red Hat Enterprise Linux, developed by CERN and Fermilab.
https://www.scientificlinux.org/
http://linux.web.cern.ch/linux/

[15] **The CMS Trigger Supervisor Project.** *"The CMS Trigger Supervisor: Control and Hardware Monitoring System of the CMS Level-1 Trigger at CERN"*. Ildefons Magrans de Abril, PhD thesis, Universitat Autònoma de Barcelona, April 2008.
http://triggersupervisor.cern.ch/

[16] **Simple Object Access Protocol (SOAP).** XML based messaging.
http://www.w3.org/TR/soap12-part1/

[17] **CMS Internal Note 2002/033.** The CMS Trigger/DAQ Group. *"CMS L1 Trigger Control System"*.

[18] **ORACLE (databases).**
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
http://www.oracle.com/index.html

[19] **Xilinx, Impact firmware loading tool:**
Xilinx, Inc.
2100 Logic Drive,
San Jose, CA 95124-3400
http://www.xilinx.com/products/design_tools/logic_design/design_entry/impact.htm

[20] *"TTCrq Manual"*. Moreira, CERN - EP/MIC, Geneva Switzerland. CERN - EP/MIC, Geneva Switzerland. Version 1.5, November 2004, linked from the QPLL website:
http://proj-qpll.web.cern.ch/proj-qpll/

[21] The CSC Track Finder Web page:
http://cern.ch/csctf
This is the home page for the group. It acts as a centralized place where various group resources can be collated.

Appendix 1. Code examples showing how to build the SPBaseInteractions objects and perform operations using the functional libraries

A.1.1 Example code showing how to get hold of a set of SPobjects using a Track Finder Crate Container object

It is suggested that you open up the *SPBaseInteractions/include/TFCrateContainer.h* interface file and quickly scan through the member functions.

This line makes a new Crate Container object and parsing the "detect" mode fills it with SPobject* corresponding to all of the live SPs detected.
TFCrateContainer TFCC_ = new TFCrateContainer("detect");*

One can query which boards are filled into the container;
TFCC_ -> listFilledBoards();
or query the instantiation of a specific one;
bool SP6_aliveOrNot = TFCC_ -> isBoardFilled(6);

One can now get hold of an SPobject in a given slot, e.g. SP slot 6;
SPobject SP_ = TFCC_ -> getSPbySlotNum(6);*
by FED-ID;
SPobject SP_ = TFCC_ -> getSPbyFEDID(890);*
or by SP number;
SPobject SP_ = TFCC_ -> getSPbySPNum(1);*
There are also TFCC_ member functions for switching between SP number, SP slot number and FED-ID.

One can alternatively get hold of a "broadcast" device which is an SPobject* that would write (but not read!) to all SPs in the crate simultaneously.
SPobject SP_br_ = TFCC -> getBroadCastObject();*
Or get hold of a pointer with which one can talk to the Clock and Control Board or Muon Sorter;
SPobject CCB_ = TFCC_ -> getCCB();*
SPobject MS_ = TFCC_ -> getMS();*
There is even a raw VME device bound to the Muon Sorter for specialist MS applications;
VMEDevice MS_Device = TFCC_ -> getMS_Device();*

A.1.2 Example code showing how to build an SPobject directly

It is recommended that one builds SPobjects using a TrackFinderCrateContainer object as in 2.1 as there are multiple checks that are performed as one does it. It is however possible to use the SPObject parser (the object used by the TFCrateContainer) directly or even just set up the constituent parts and then call the SPobject constructor directly.

Using an SPObjectParser to build an SPobject;
SPObjectParser SPObjectParser_ = new SPObjectParser();*// we could have parsed a mode here, see Table 1.
SPobject SP_ = SPObjectParser_ -> getSPObject(int slotNumber);*

Building an SPobject directly (this is the guts of what the SPObjectParser does);
string AddressTable = "AddressTable_SP2002.xml"; //The full path to the xml table containing the register base offsets.
HAL::HALUtilities HALUtil;
unsigned long VMEbase = HALUtil.slot2address(VMEslot);
HAL::VMEAddressTableXMLFileReader addressTableReader(AddressTable.c_str());
HAL::VMEAddressTable addressTable("SP address table", addressTableReader);
BUSADAPTER busAdapter(CAENLinuxBusAdapter::V2718,adapternumber);
*SPobject *SP = new SPobject(addressTable, busAdapter, VMEbase);*

One can similarly construct raw VME devices for the MS or SP broadcast devices using slot 30 as an input.

A.1.3 Example code demonstrating how to use the SPobjects

Once one has obtained an SPobject* (SP_) using one of the methods above, then one can use it to talk to the electronics and perform functions.

One can query the SPobject itself for certain attributes of the Sector Processor such as the firmware dates or its own name;
SP_ -> FWDate("DD");// Return the firmware date of the DD chip
SP_ -> whoAmI(); // Returns the name of the board (SP1 to SP12, CCB, MS etc.).

One can carry out simple register reads and writes. The first step is to get hold of the FPGAobject corresponding to the chip one wants to talk to. For the SP, one can interact with the main logic chip on the mezzanine card, "SP", the VME interface chip, "VM", the trigger data chip, "DD" or one of the 5 front FPGA chips, "F1" to "F5". One can also parse "FA" which would return a pointer to all front FPGAs. One could only use the "FA" object to make a write. Incorrectly trying to read from "FA" would cause an error.

Once one has got hold of the FPGAobject then one can make a read from it or a write to it by calling the read or write functions. Both take 2 arguments of register and "muon number". The registers are described by the Backplane Interfaces document linked from [8]. The Muon Number refers to the case where there are 3 muon lines for some registers. One has the choice to parse "M1" to "M3" or alternatively "MA" representing all lines. As with "FA", one can make a write to "MA" which would result in the same effect as making 3 individual calls to M1, M2, M3. Also, as with FA, one may not make simultaneous reads from all 3 lines using MA. This has an exception in that there are not necessarily 3 register lines for all registers and in the case of these registers one should use MA both for reading and writing.

e.g. make a write of 0x1 to the SP-chip, the CSR_LNK register;
SP_ -> getFPGA("SP") -> write("CSR_LNK","MA", 0x1); //make a write to register
make a read from the trigger primitive ("valid pattern") counter of the 2nd muon (trigger) link of front FPGA 4;
int count = SP_ getFPGA("F4") -> read("DAT_VPC", "M2"); //make a read from a register.

It is possible to use the higher level functions available from the other base libraries as described in Section 3. Typically one constructs a simple class of type corresponding to the required function and then parses the SPobject* to it so that it can make a series of reads/writes to the hardware.

e.g. Use the *SPFunctions/timings* class to run the function (int *readAFD_FA(SPobject*)*) that returns the AFD value and checks that this value is common to all front FPGAs (FA);
timings* timings_ = new timings();
int AFDNumber = timings_ -> readAFD_FA(SP_);//use the timings_ object to run the function.
Of course, if one is using a TFCrateContainer then one can access the SPobject directly in the argument;
int AFDNumber = timings -> readAFD_FA(TFCC_ -> getSPbySPNumber(1));

Appendix 2 : Track types by mode table.

This table shows the track mode assigned by the SP core on the basis of the constituent inter station extrapolation(s).

Table 8. Track Modes assigned by the SP as a function of the extrapolations. This table is courtesy of Alex Madorsky (University of Florida).

Mode	Rank	Track asssemler key station 2	Track assembler key station 3	Track assembler key station 2 to DT
0	no track			
1	bad phi road			
2	12	1(1)-2-3	1(1)-2-3	
	17	1(2)-2-3	1(2)-2-3	
	1c	1(3)-2-3	1(3)-2-3	
	1f	1(1)-2-3-4	1(1)-2-3-4	
	21	1(2)-2-3-4	1(2)-2-3-4	
	23	1(3)-2-3-4	1(3)-2-3-4	
3	11	1(1)-2---4		
	16	1(2)-2---4		
	1b	1(3)-2---4		
4	10		1(1)---3-4	
	15		1(2)---3-4	
	1a		1(3)---3-4	
5	4	2-3-4	2-3-4	
6	6	1(1)-2		1-2
	b	1(2)-2		
	e	1(3)-2		
7	5		1(1)---3	
	a		1(2)---3	
	d		1(3)---3	
8	3	2-3	2-3	
9	2	2-4		
a	1		3-4	
b	unused			
c	13			1-2-b1(1)
	18			1-2-b1(2)
	1d			1-2-b1(3)
d	unused			
e	9			2-b1(1)
	c			2-b1(2)
	f			2-b1(3)
f	halo trigger (since 2007)			

Explanation:
Track assembler columns show which extrapolations are used in a track.
Notation:
- numbers 1,2,3,4 mean ME stations 1,2,3,4
- b1 means DT station 1
- 1(2) means extrapolation quality to station 1 is 2.

Example:
if "1(2)-2-3-4" appears in "Track assembler key station 2" column, it means a track was built from the following extrapolations:
- ME1 to ME2 (quality 2)
- ME2 to ME3
- ME2 to ME4

Appendix 3. Installation of *TriDAS/trigger/csctf*

This section gives details on the installation of the Track Finder online software. It is clear that over time dependencies and exact paths will vary as the library linking and make file setup change, however, it is nonetheless thought useful to include some of the finer details as they really are now (2008) below.

The first step is to set up a linux machine (we use SLC[14]) with XDAQ[11] installed and ensure that there is access to the CVS TriDAS repository so that the code can be checked out.

edit .bashrc so as to include the lines:
source /home/<mypath...>/TriDAS/trigger/ts/toolbox/scripts/environment.sh
export CVSROOT=:kserver:isscvs.cern.ch:/local/reps/tridas
export CSCTF="/home/<mypath...>/TriDAS/trigger/csctf"

The next step is to check out from CERN CVS[9] the packages *TriDAS/trigger/csctf* and *TriDAS/trigger/ts.* One can browse the CVS code and directories from the web link given in [9].

cvs co TriDAS/trigger/csctf
cvs co TriDAS/trigger/ts

edit TriDAS/trigger/ts/toolbox/environment.sh so that the local path lines match the correct local path. i.e. The first line becomes: *export BUILD_HOME=/home/<mypath...>/TriDAS*

Make the base library;
cd SPBaseInteractions
make

Ensure that the correct paths to the folders where the dynamically bound libraries are being created exist in the environmental variable *LD_LIBRARY_PATH*. i.e. The compile above should have created the library *CSCTF/SPBaseInteractions/lib/linux/x86_slc4/SPBaseInteractions.so.* The path to the folder containing it, x86_slc4, should appear in the output of *echo LD_LIBRARY_PATH*. If it does not then it has to be added to it before one can use it in further compilations.[12] The line to add it in a bash environment would be;

export LD_LIBRARY_PATH=$CSCTF/SPBaseInteractions/lib/linux/x86_slc4:$LD_LIBRARY_PATH

This could be added to the *.bashrc* which would mean it would be automatically added for future logins. All of the compiled .so libraries should similarly appear in the *LD_LIBRARY_PATH*.

make the SPFunctions library and its test executables
cd SPFunctions
make
cd test
make bin

make the LUT and FW library with its test executables:
cd SPLUTsAndFirmWare
make
cd test
make bin

make the rootUnpackSPDAQ executables: Make sure that ROOT[6] is correctly installed on the system by typing in root and seeing it start, usually with the ROOT flash screen.
cd rootUnpackSPDAQ
./compile

make the SPValidation self -test library with its test executables:
cd SPValidation
make
cd test
make bin

---

[12] At one time, the group used a strategy of a linked library list which would automatically be sourced and add pertinent directories to the users path. This was replaced by the trigger supervisor *environment.sh* list, which, unfortunately is currently notoriously unreliable. This is why the user may have to worry about these standard compilation linking jobs.

make the stand alone executables, there is no library:
cd SPStandalones/test
make bin

make the *CSCTF/ts* Trigger Supervisor library:
cd ts
make

Appendix 4. SBS Bus Adapters

As of April 2007, the support of SBS controllers as standard was deprecated within the CSCTF online software since the default controller was the CAEN[12]. Nevertheless, resurrection of a test stand that used an SBS to control the Track Finder Crate would not be too difficult since many of the SBS hooks in the code have been left in as comments. From 2007 onwards, much of the SBS driver code is supported within XDAQ, so depending on the exact setup, it may be possible to skip

1/ Get the SBS working;

install the PCI card and some driver libraries; we used to have the SBS directories in
home/csctf/VMEController (or something of type /home/usr/VMEController) and the commented out settings in the .bashrc,
Makefiles etc will reflect this.

2/ put back in the commented out lines (find dch, sbs).. in:

bashrc; put back in the lines for variables VME_DIR and SBS_VERSION
if you changed names/locations/version in (1) then this will need some attention

config/compdef.mk (really nasty little file)

SPBaseInteractions/Makefile
SPBaseInteractions/include/SPObjectParser.h
SPBaseInteractions/src/common/SPObjectParser.cc

Finally, worry about the compiling the packages below as you need them;

SPFunctions/Makefile

SPLUTsAndFirmware/Makefile

SPStandAlones/test/GNUMakefile
SPStandAlones/test/readWriteRegisterFuncts/*.cpp as needed, both includes and actual instatiation of bus adapter

SPValidation/Makefile
SPValidation/test/*.cpp as needed

TFHyperDAQ/Makefile