ALPHA DATA
BEAM

CERN Trajectory Measurement System

Software Manual

Project	CERN-TMS
Date	2013-02-01
Reference	Cern-tms/TmsSoftware
Version	2.0.0
Author	Dr Terry Barnaby

Table of Contents

1. References

- IT-3384/AB: Technical Specification for a new trajectory measurement system for the CERN Proton Synchrotron.
- TMS design documents systemDesign, pupeFpgaDesign, pupeBoardDesign.
- TMS Development and Support website at: https://portal.beam.ltd.uk/support/cern/.

2. Introduction

This TMS Software Manual provides an overview of the CERN Trajectory Measurement System (TMS) Software and describes in detail the configuration and usage of the system. It is mainly concerned with the usage of the Software API's for controlling and gathering data from the CERN Trajectory Measurement System (TMS).

Other documents available include:

- The “TMS Overview” document that provides an overview of the system.
- The “TMS Maintenance” document which describes system installation and maintenance.
- The” TMS Test” document describing system testing.
- The “TMS SigGen” document describing the test signal generator.
- The “TmsControlGui” document describes the TmsControlGui test and diagnostics GUI applicatrion.
- The “TMS API” documentation describing, in detail, the TMS software API's.

The TMS system is designed to measure the trajectory of particle beam's within the CERN Proton Synchrotron and Booster Proton Synchrotron machines. It is able to measure the amplitude and x/y displacement of the individual particle bunches as they pass each of the analogue sensors in the ring. The system integrates the amplitude of the data received for each particle bunch and stores the results in memory for later data access. In order to accurately measure the particle bunches the system uses phase locked loops to synchronise the data capture to the incoming data.

The system continuously samples over 120 Analogue channels at 125MHz, 14 bits and processes this data in real-time to determine information on the position of particle bunches as they orbit at around 437kHz. The system thus captures and processes over 15 billion samples per second. Multiple Xilinx Vertex 4 FPGA's are employed in a modular system to capture and process the data. The system is controlled over a Gigabit Ethernet network from which portions of the resulting data can be accessed.

Within the TMS, the main, high speed, data processing work is carried out in FPGA hardware using specially developed FPGA firmware written in VHDL. The system software’s main responsibility is to provide control, timing, data access and test functions for the system.

To gain an understanding of the overall systems design please refer to the “TMS Overview” document.

3. System Overview

The main, high speed, data processing work is carried out in FPGA hardware. The system software’s main responsibilities are to provide control, data access and test functions for the system.

All of the system software is based on the Linux operating system. This provides a reliable and flexible system that can be easily maintained locally and remotely. All of the software is Open Source and thus all

source code is available.

All communications with external systems is through the System Controller (SC) which supports a simple API to control and gather data from the system. The System Controller interrogates the individual Pick-Up Processing Engines (PUPE) via the local Gigabit Ethernet network and the Module Controllers (MC). The TMS's API can be used across the network interface from a remote system or locally from applications running on the System Controller.

The System Controller also supports a basic HTTP web interface for viewing system status and getting data.

The software has been developed on the GNU/Linux operating system using the Open Source GNU tool-set. The software is predominantly written in the 'C++' language.

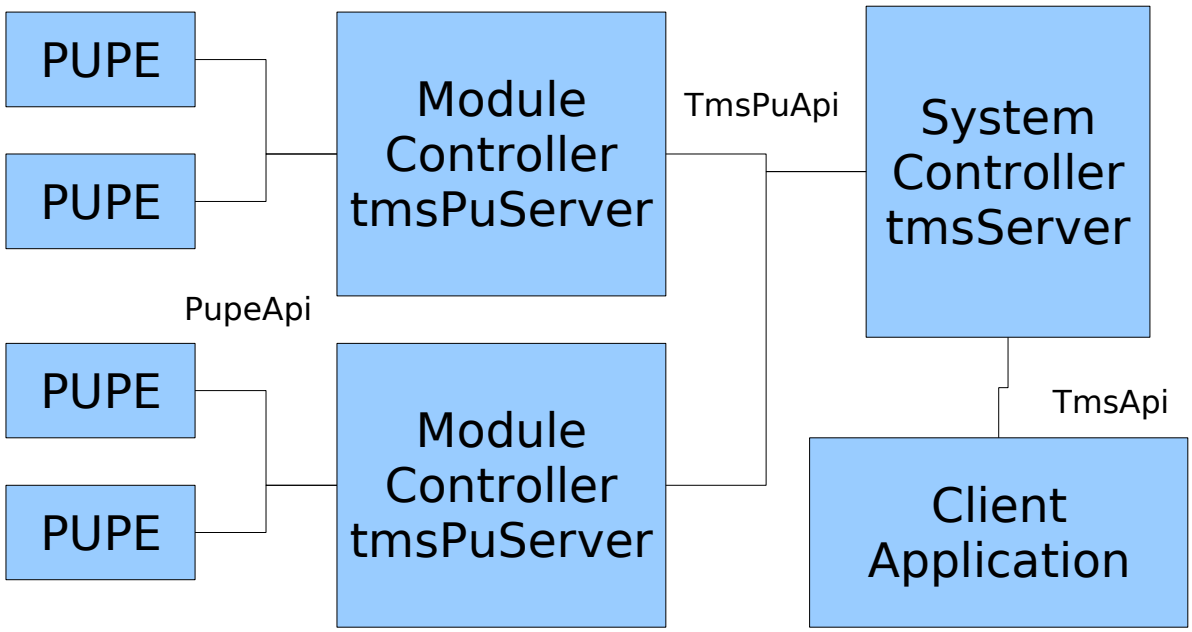

Illustration 1: TMS Main System Modules

From the software's perspective there are four main modules in the system, the Pick-Up Processing Engine (PUPE), the Module Controller (MC), the System Controller (SC) and the Client Application (CLIENT).

The TMS has a private Gigabit Ethernet Network to which the Module Controllers and System Controllers are connected. The System Controllers have dual Gigabit Ethernet interfaces, one is connected to the TMS's private network and the second is connected to the CERN local area network.

3.1. Pick-Up Processing Engine (PUPE)

The PUPE is the main module in the TMS system. It performs the analogue data capture and real-time data processing functions of the TMS. The PUPE is based on FPGA technology and is implemented as a cPCI board installed in a Compact PCI 19inch rack. Each PUPE engine implements 3 pick-up processing channels each having 3 ADC's. The PUPE is accessed via the cPCI bus from a cPCI Module Controller board.

The PUPE FPGA boards are booted from the systems Module Controller (MC) using the standard Alpha

Data FPGA boot protocol. Control and data access is implemented using the PUPE API across the 64 bit 33MHz cPCI bus. The PUPE API is defined in the PupeFpgaDesign document.

3.2. Module Controller (MC)

The module controller is a conventional cPCI system controller. It is based on an Intel x86 CPU and has some boot FLASH memory, at least 1 Gigabyte of RAM, a cPCI bus interface and three Gigabit Ethernet ports. The boards used include a Concurrent Technologies PP 41x/03x board and a PP 712/083-13.

The Module Controller boot's from the main System Controller (SC) over a Gigabit Ethernet interface and runs a small Linux based operating system. It is responsible for booting and managing the PU processing engines (Up to 18 Proton Synchrotron PU's) on its cPCI bus. Communications between the SC and the individual PU processing engines is also be handled.

The Module Controller runs the **tmsPuServer** program that implements a simple network based API, TmsPuApi, for control and access to the individual PUPE channels.

3.3. System Controller (SC)

The system controller is a standard Intel Xeon based server computer system. It is housed in a separate 2U or 4U 19" rack enclosure. The system controller has at least 2 Gigabyte's of RAM and dual SATA disk drives in a RAID 1 configuration for disk redundancy. These disks contain all of the TMS's software, FPGA firmware and configuration information. The controller has dual Gigabit Ethernet interfaces, one connected to the Gigabit switch that communicates with the Module Controllers and one connected to the sites LAN for remote access to the system.

The SC can run without a monitor, keyboard or mouse, but these can be connected if desired to see diagnostics locally. All system configuration and maintenance can be carried out over the Ethernet network either through Linux or at a BIOS level through the IPMI interface. The system controller runs the Linux operating system.

As well as providing a control and data interface to the Trajectory Measurement System, the software on the system controller implement's system boot, system configuration, system test and fault diagnostics functions. Access to this information is made available to operators via a web based interface as well as through command line and X-Windows GUI API's.

3.4. Client Application's (CLIENT)

The client applications are CERN's system control and data gathering applications. These will probably reside on different systems and communicate with the TMS through the Gigabit Ethernet interface. It is also possible for CERN to implement these applications on the TMS's System Controllers if desired. The Client Applications will translate between CERN's specific control and data access protocols and the TMS's internal control and data protocols.

4. Software Environment and Tools

All of the TMS's software is developed within the Linux operating system environment and uses the Linux operating system as its base system layer. The base Linux distribution used is Centos 6.3. The TMS individual systems, the MC and SC, have a limited installation of this operating system, the Module Controllers in particular have a very cut down system based on the Busybox utility.

The main software development tools are the GNU tool-set and the main development language is 'C++'. These tools are installed on the System Controllers but can be used on a separate systems. The SVN

version control system is used for version control.

All software is supplied in source code as well as binary forms.

5. Operational Overview

The TMS system is designed to measure the trajectory of particle beam's within the CERN Proton Synchrotron. It is able to measure the amplitude and x/y displacement of the individual particle bunches as they pass each of the analogue sensors in the ring. The system integrates the data received for each particle bunch and stores the results in PUPE memory for later data access. In order to accurately measure the particle bunches the system uses phase locked loops to synchronise the data capture to the incoming data.

The TMS system has been designed to work with single ring or multi-ring systems such as the Booster PS machine.

The TMS operates in processing cycles lasting about 1.2 seconds. A number of hardware timing signals are provided by the PS machine to synchronise the TMS to various PS machine events. The timing signals are:

Name	***Description***
10 MHz system clock	Master system clock. The ADC's 125 Mhz sampling clock is optionally synchronised to this clock and all of the digital timing signals, except the Injection signal, are re-synchronised to the +ve edge of this clock within each FPGA.
FRef Input	PS machine reference frequency. This is a square wave signal synchronised with the particle beams orbit at one point in the ring.
CYCLE_START	Start of a machine cycle. The PUPE's cycle time ms counter is reset to 0 and the system starts capturing new set of cycle data.
CYCLE_STOP	End of a machine cycle. Data from the previous cycle will now be made available to client applications.
CAL_START	Start of calibration period.
CAL_STOP	End of calibration period.
INJECTION	Injection. The particle beam has been injected into the PS ring.
H-CHANGE Input	Harmonic change. The PS ring moves from operating at one harmonic to another.

As well as the hardware timing signals there is one software timing signal. The PS system will tell the TMS the cycle number and cycle type of the next processing cycle prior to the hardware CYCLE_START signal. This is handled by calling the **setNextCycle** function of the TMS API with the cycle number and cycle type. This should be done at least 10ms before the CYCLE_START signal for the cycle it refers to. An ideal timing would be on the CYCLE_STOP event for the previous cycle. This gives the system plenty of time to load the FPGA's with the new state/phase tables.

The cycle number is a 32bit unsigned incrementing integer and the cycle type an ASCII string. The cycle type string defines the PUPE state/phase tables to be used for measuring the particle beam that will be present in the machine for that cycle. The TMS will arrange for the correct State/Phase tables to be loaded

before the CYCLE_START event for the next processing cycle.

The TMS system will continually capture, process and store the resulting data into the individual PU channel memory on the FPGA's. There is sufficient memory to store about 3 cycles worth of data in this memory depending the the particle beam's harmonic number and number of bunches. Client applications can read this data within the, around, 2.4 second window that the data is available. As well as the main particle beam's Sigma/DeltaX and DeltaY integral data, the TMS system computes, on the fly, the average Sigma/DeltaX and DeltaY values over each1ms period. This, lower bandwidth, data can be accessed by the clients also within the same 2.4 second window that the data is available.

5.1. TMS Cycle Parameters State/Phase Tables

Each PUPE channel has an individual state table and PLL phase table system that is used to manage inter-cycle events and provide the necessary phase locked timing signals for data capture and processing of a particle beam's trajectory. The TMS keeps a library of these Cycle Parameter State/Phase tables indexed by a cycle type string, ring number and channel number. The library is stored on disk as a set of simple ASCII files. See the section on Cycle Parameter file format for more details. The master **TmsServer** program sends the complete set of Cycle Parameter data to each **TmsPuServer** program on initialisation. Thus each Module Controller has the complete library of State/Phase tables, in internal data structures, so they can be loaded into individual PU channels on the fly. The **TmsPuServer** program will load each FPGA's PUPE channel with the cycle parameters on thesetNextCycle API call or on the CYCLE_STOP event which ever is the later.

The Cycle Parameter information is normally the same for every ring and every channel within the ring except for and individual frefPhaseDelay parameter which defines the position of the pick-up within the PS ring. It is, however, possible to load an individual set of Cycle Parameters into an individual PUPE pick-up channel if necessary on a ring and/or channel number basis.

The **TmsServer's** TmsControl API provides the ability to update the Cycle Parameter library while the system is in operation.

The set of Cycle Parameters will need to be set up for all of the possible PS machine beam type cycles. We have supplied a simple program, **tmsStateGen**, that will generate simple State/Phase tables for some test beam sources. It is also possible to use the **tmsControlGui** application that has a basic Cycle Parameter editor built in. Information on the Cycle Parameters and their use is given below. Full details of the content of the State/Phase tables can be found in the **TmsPupeFirmware** manual.

5.1.1. Cycle Parameter State/Phase Table Configuration

Most of the Cycle Parameter information is loaded into the FPGA's individual PUPE channel's registers and table RAM. However, some of the parameters are used for the software when retuning information on the cycle and for fetching data from the PUPE memory. The Cycle Parameter information contains the following parameters:

Field	*Description*
cycleType	This is the Cycle Type string this configuration is for.
name	A name for this table. Used for debugging individual ring/channel tables
info	An arbitrary string describing the BEAM type this configuration is for.
ring	The ring number this configuration is for. A ring number of 0 indicates all rings.

channel	The channel number this configuration is for. A channel number of 0 indicates all channels.
pllCycleStartFrequency	This defines the initial PLL frequency. This is loaded on the START_CYCLE event. This value should be close to 2^32 * FREF / 125.0e6.
pllInitialFrequency	The initial value loaded into the PLL's initial frequency register after the pllInitialFrequencyDelay period from CYCLE_START. This value should be close to 2^32 * FREF / 125.0e6.
pllInitialFrequencyDelay	This is the delay, in milliseconds, after the CYCLE_START event that the pllInitialFrequency value is loaded into the PLL. The delay is there to allow the FREF timing input to have stabilised before the PLL attempts a lock.
pllFrefGain	This is the gain value applied to the incoming FREF signal before using as a reference for the PLL. FREF is a binary timing signal. The FREF signal used for the PLL will have the values +pllFrefGain and -pllFrefGain. A typical value for this would be around 4096 to match the incoming values of Sigma.
pllGain	This parameter provides a control of the gain of the PLL feedback path. Its value defines the number of right shifts (divides) that are applied to the binary error value. The PLL filters have a gain of about 128, so a value of 7 here will be equivalent of a loop feedback gain of 1. Note that the PLL feedback gain is also dependant on the level of Sigma and pllFrefGain depending the the reference source being used.
pllDdsMinimum	This defines the minimum frequency that the PLL's frequency register will go down to. If this value and pllDdsMaximum are set to 0 there are no bounds to the PLL frequency.
pllDdsMaximum	This defines the maximum frequency that the PLL's frequency register will go up to. If this value and pllDdsMinimum are set to 0 there are no bounds to the PLL frequency.
settings	This is an array of strings describing the basic parameters for each of the TMS's data capture states. It is used to allow Cycle Information editors to edit the State/Phase tables, contained within the Cycle Information, at a high level.
frefPhaseDelay	This is an array of phase shift values for each of the PU channels. The phase shift effectively applied to the incoming FREF global timing signal so that the locally generated FREF signal and other PLL signals can be phase aligned to the channels PS ring position. Its value can be plus or minus and is in 1/512 of a rotation or 0.703125 degrees.
stateTable	This is the array of state table entries. There can be up to 14 entries in this table. Each entry defines a set of events that will move the pick-up channel to a new state, defines the settings of various parameters for the state, the number of particle bunches that will be acquired and the Phase Table to be used.

State Table Configuration

Field	*Description*
period	The Cycle period this state is used for
state	This defines the state configuration bits and the state transition on event values. It is a 32bit value with the bits defined below.
numBunches	This defines the number of particle bunches that will be acquired per orbit. It is used for the calculation of a data offset based on the orbit number.
harmonic	This is the machines harmonic number
bunchMask	Bitmask defining which buckets the bunches are captured from. Bit 0 is bucket 1, bit 1 is bucket 2 etc
phaseTable	This is the phase table to be used for the state. There are 512 8bit entries in the table. The definition of the bits is given below:

State Bit Definitions

Name	*Bits*	*Description*
acquireData	0	The system will perform data acquisition if this bit is set. This will allow entries to be made in the CycleTimingTable and in the CycleDataTable if appropriate GATE strobes are present in the PhaseTable for this state.
pllReference1	1	Selects the PLL source for the Filter 1 path. 0 – selects FREF, 1 – selects Sigma.
pllReference2	2	Selects the PLL source for the Filter 2 path. 0 – selects FREF, 1 – selects Sigma.
pllFeedbackSelect	3	Selects which of the Filter outputs to be used for the PLL error value. 0 – selects filter 1, 1 – selects filter 2.
pllLO1FromAddress	4	This selects which PLL signal is to be used as the PLL internally generated FREF for the Filter 1 path. 0 selects the LO1 phaseTable bit, 1 – selects the MSB of the PLL's phase counter.
pllLO2FromAddress	5	This selects which PLL signal is to be used as the PLL internally generated FREF for the Filter 2 path. 0 selects the LO2 phaseTable bit, 1 – selects the MSB of the PLL's phase counter.
cycleStop	11:8	This defines which state to move to when a CYCLE_STOP event occurs.
calStop	15:12	This defines which state to move to when a CAL_STOP event occurs.
calStart	19:16	This defines which state to move to when a CAL_START event occurs.
injection	23:20	This defines which state to move to when a INJECTION event occurs.
hchange	27:24	This defines which state to move to when a HCHANGE event occurs.

delay	31:28	This defines which state to move to 16 FREF periods later. It can be used to add a delay to the state/phase table switch or add a section of different state/phase table settings for a 16 FREF period after an event.

Phase Table Bit Definitions

Name	*Bits*	*Description*
lo1	0	The LO1 PLL signal used in the filter 1 feedback path.
blr	1	The base line restoration signal. When high BLR is being calculated
gate	2	The gate signal used to acquire data.
lo2	3	The LO2 PLL signal used in the filter 2 feedback path.
meanFilter1	6	A pulse which sends the current integral values into the bunch mean filter 1. Typically used to return integral values every ms for all particle bunches.
meanFilter2	7	A pulse which sends the current integral values into the bunch mean filter 2. Typically used to return integral values every ms for the first particle bunch.

5.1.2. Cycle Parameter File Format

The State/Phase table library files have a simple ASCII format. There is a single entry per line consisting of a field name followed by a colon and then the data value. The field names match the State/Phase table parameter names appending with an integer index in the case of arrays.

The Cycle Parameter files are read when the **TmsServer** program is started or when the API function **init** is called. Normally the TMS API function **setControlInfo** is used to update an individual set of Cycle Parameters. When this function is used the Cycle Parameter files on the server are updated and the Cycle Parameters are propagated to all of the TMS module controllers.

5.1.3. Cycle Parameter Configuration

The Cycle Parameters can be configured in a number of ways. The TMS software provides a simple high level configuration system that can be accessed from the **tmsStateGen** command line program or the **tmsControlGui** GUI control program. The user can also generate the Cycle Parameters manually into an ASCII file and upload them to the TMS using the **setControlInfo** API function via the **tmsControlGui** or **tmsControl** programs.

The TMS configuration system is implemented in the TMS API library in the **CycleParamEdit** class. This simple software class provides the ability to read and write the Cycle Parameter files and to setup the parameters based on a high level description of the PS cycle. It is used by the **tmsStateGen** and the **tmsControlGui** programs.

In the TMS each PS cycle is split into separate active cycle periods. These periods are defined by events such as CYCLE_START, INJECTION and harmonic change. The high level cycle description defines all of the fixed cycle parameters and then has a list of parameters for each of these active cycle periods. For each cycle period the following parameters are defined:

period	The cycle period
state	Defines next state numbers on events and the bit field settings for the state.
bunchMask	The set of bunches to capture bit mask. This defines the bucket to bunch relationship. If a bit is set then the bunch data will be captured for the given bucket. Bit 0 is for bucket 1, Bit 1 is for bucket 2 etc.
mean1Mask	The set of bunches to pass through meanFilter1. This is used as the mean filter strobe for the new bunch mean system as well as then older all bunch average system. It should generally have the same value as the bunchMask.
mean2Mask	The set of bunches to pass through meanFilter2. Note that this function is depreciated in favour of the new bunch mean system.
lo1Harmonic	The LO1 harmonic number used in this state. This defines the number of buckets.
lo1Phase	The phase offset of the LO1 as a fraction of FREF (+-1.0).
lo2Harmonic	The LO2 harmonic number used in this state. This defines the number of buckets.
lo2Phase	The phase offset of the LO2 as a fraction of FREF (+-1.0).
gateWidth	The gate pulse width as a fraction of LO (0 - 1.0)
gatePhase	The gate phase offset as a fraction of LO (0 - 1.0)
blrWidth	The gate pulse width as a fraction of LO (0 - 1.0)
blrPhase	The gate phase offset as a fraction of LO (0 - 1.0)

The TMS PUPE's PLL has two separate frequency generators LO1 and LO2 complete with separate feedback filters. Normally these are used in tandem so that at an event the system can swap its lock frequency with the minimum of phase jitter. Given a set of cycle period parameters as defined above, the software will then generate a set of state/phase tables with appropriate alternating LO1 and LO2 entries. So, for example, with a BEAM that may have a calibration period and then a harmonic 8 period after injection followed by a harmonic change to harmonic 16, you just enter the details for each of these periods. The software would then generate the necessary PUPE states as follows:

State	***Description***
0	Awaiting calibration or injection LO1 – FREF, LO2 - Harmonic 8 settings, Active – L01
1	Calibration period, awaiting calibration stop or cycle stop LO1 – FREF, LO2 - Harmonic 8 settings, Active – L02
2	Awaiting injection LO1 – FREF, LO2 - Harmonic 8 settings, Active – L01
3	Event0 period, harmonic 8 capture LO1 - Harmonic 16 settings, LO2 - Harmonic 8 settings, Active – L02
4	Event1 period, harmonic 16 capture LO1 - Harmonic 16 settings, LO2 - Harmonic 8 settings, Active - L01

So the states toggle between using LO1 and LO2 and corresponding LO phase tables are set-up for the current and next state. Note the the FREF from LO1 or LO2 is not actually used as the PLL's address MSB is used for this so that the phase delay per channel works (assuming this option is enabled which it is by default).

5.1.4. Cycle Parameter Configuration Notes

The PUPE FPGA's state/phase table system has many possible settings. The TMS software requires some of these settings to be configured in a certain way for the current software to work correctly. Some notes are this follow:

- In the switch table the ACQ bit should be set for all cycle periods. This is because the software assumes a continuous set of timing data in the cycle timing table.
- If any of the states is set to go to error state then the data capture for that cycle will be aborted and an appropriate error returned to all clients requesting data for that cycle.
- The current software is designed to assume the MSB of the PLL is used as the local FREF signal rather than the appropriate LO1 or LO2 frequency generators. This allows a single set of state/phase tables to be used in all channels with a simple phase delay parameter to define the ring position.
- The Mean Filter Clock Enable 0 phase table bit is used to sample the bunches for the new bunch mean filter system as well as the old bunch filter 0 system.
- When using the TmsControlDiagnostics to look at the PLL timing with respect to Sigma, note that the FPGA will delay sigma by approximately two clock cycles prior to the integration stage. This the GATE, BLR, Mean1 and Mean1 filter pulses should be delayed with respect to the Sigma signal displayed.

5.2. TMS Data Access

The TmsServer's TmsProcess API provides the ability for client applications to easily obtain the trajectory data from a PS machine cycle. There is a TmsServer program for each ring in the PS system. The API provides a getData() call that takes a DataInfo parameter structure defining the data required. The contents of this DataInfo parameter are as follows:

Field	***Description***
cycleNumber	The PS Cycle number to fetch data from.
channel	The pick-up channel number.
cyclePeriod	The cycle period the data to fetch data from.
startTime	The start time in milli-seconds from the start of the required Cycle Period.
orbitNumber	The starting orbit number (starting from 0).
bunchNumber	The bunch number (starting from 1 (0 is all bunches)).
function	The data processing function to perform or performed.
argument	The Argument to the data processing function.
numValues	The total number of data points to return.

The **cyclePeriod** value defines which particular period from the cycle to fetch data from. Each TMS processing cycle is split into the following cycle periods:

Period	*Description*
CyclePeriodStart	From the start of the cycle
CyclePeriodCalibration	From the start of the calibration period. Ie after CAL_START.
CyclePeriodEvent0	From the data after INJECTION to the first harmonic change
CyclePeriodEvent1	From the data after the first harmonic change to next harmonic change
CyclePeriodEvent2	From the data after the second harmonic change to next harmonic change
CyclePeriodEvent3	From the data after the third harmonic change to next harmonic change
CyclePeriodEvent4	From the data after the forth harmonic change to next harmonic change
CyclePeriodEvent5	From the data after the fifth harmonic change to next harmonic change
CyclePeriodEvent6	From the data after the sixth harmonic change to next harmonic change
CyclePeriodEvent7	From the data after the seventh harmonic change to next harmonic change

The functions currently defined are as follows:

Function	*Description*
DataFunctionRaw	The raw Sigma,DeltaX,DeltaY integrated data
DataFunctionMean	The mean Sigma, DeltaX, DeltaY integrated data over 1Ms sample periods. The mean values are available for all bunches on all channels.
DataFunctionMeanAll	The overal mean Sigma, DeltaX, DeltaY integrated data over 1Ms sample periods for all bunchens. The mean values are available for all channels.
DataFunctionMean0	The mean Sigma,DeltaX,DeltaY integrated data. 1Ms sample period for all bunches although this is programmable. This function is depreciated in favour of the new DataFunctionMeanAll function.
DataFunctionMean1	The mean Sigma,DeltaX,DeltaY integrated data. 1Ms sample period for bunch 1 although this is programmable. This function is depreciated in favour of the new DataFunctionMean function.

When a client calls the getData() call it will be blocked until the data for the cycle requested becomes available. The getData() call can return a number of possible errors. See the detailed API documentation and the section on Error Handling for more details.

It is also possible to use the TMS HTTP web interface to access the data although as this is an ASCII based method, it is a slower interface.

6. Software Structure

There are two main software programs that run on the TMS System: The TmsServer and the TmsPuServer programs. The TmsPuServer program runs on the Module Controllers and is responsible for managing the PUPE engines. It uses the **PupeApi** to communicate with the individual Pick-Up processing

engines and implements the TmsPuControl and TmsPuService API's. The TmsServer program runs on the System Controller and provides overall control of the system. There is one TmsServer processes for each ring of the system. They use the TmsPuControl and TmsPuService API's to communicate with the TmsPuServer programs and implement the TmsControl and TmsService API's.

These programs are automatically started at boot time and run with a real-time process priority. They are multi-threaded programs and make use of the multiple CPU cores present on both the Module Controllers and System Controller.

There are also two test and control applications. These are called tmsControl and tmsControlGui. The tmsControl application is a simple command line application that can control and read data from the TMS system. The tmsControlGui application is a simple GUI test application that performs the same role as the tmsControl application but has a GUI for control, diagnostics and the displaying of results. These are documented in the TmsTesting and TmsControlGui documents.

To aid with performing system tests there are two utility applications. The tmsSigGen program is designed to produce simple test signals emulating BEAM types that could be present in the PS machine. It can be used to generate Test waveform files for directly loading into the PUPE's test data RAM or can be used to drive the Tms Signal Generator AWG test board to produce simulated data and timing signals for the system.

The tmsStateGen program is designed to produce simple PUPE Cycle Parameter State/Phase tables for the example PS BEAM sources. The TmsControlGui application can also be used to generate simple Cycle Parameter State/Phase tables.

The System Controller is responsible for:

1. Storing all of the systems software and FPGA firmware.
2. Managing software version control and software/firmware packaging by using the RPM packaging format.
3. Providing system logs from all components including the Module Controllers. All of the Module Controllers send their logs to this system.
4. Providing Network information to the Module controllers through DHCP.
5. Providing the Module controllers with their kernel and root file systems through NFS.
6. Providing time information to the Module Controllers.
7. Providing a diagnostics login to the Module Controllers.
8. Providing an effective fire-wall for the Module Controllers.
9. Managing the TMS's internal network bandwidth.
10. Providing the future ability to post-process data.
11. Providing a TMS GUI Diagnostics and Control interface.
12. Providing a TMS Web interface.
13. Providing a software development environment for the TMS system.

The TmsServer program is responsible for:

1. Overall TMS system management and control.
2. Maintaining the Cycle Parameter State/Phase table database. (Master TmsServer only)
3. Managing the sending of appropriate Cycle Parameter State/Phase tables to the TmsPuServer's. (Master TmsServer only)
4. Making sure that the setNextCycle() information is sent through on time. (Prioritises system tasks and internal network bandwidth).

5. Providing a queue for client requests and prioritises these queues.
6. Providing system events to the client applications.
7. Providing the ability to map individual PU channels to particular physical PUPE channels.
8. Managing the TmsPuServer programs.
9. Providing overall system error handling and collating errors to be logged or sent back to the client applications.
10. Providing overall system state information.
11. Providing overall system statistics information.
12. Providing system test facilities.
13. Providing background diagnostics functions.
14. Providing the ability to perform whole system measurements from the whole set of PUPE's for measurements such as phase space plots.
15. Providing the future ability to post-process data.
16. Providing the future ability to cache data requests to increase performance with multiple clients.

The TmsPuServer program is responsible for:

1. PUPE Board management and control.
2. Loading the FPGA's with appropriate bitfile firmware.
3. Loading the FPGA's with the appropriate Cycle Parameter State/Phase tables.
4. Receiving timing events from the PUPE boards.
5. Generating simulated timing signals.
6. Providing Module state information.
7. Providing Module statistics information.
8. Providing Module test facilities.
9. Read the raw and mean data from the PUPE boards.
10. Providing the future ability to post-process data.

6.1. File Structure

Most of the TMS software is installed in /usr/tms this has the following sub directories:

Directory	*Usage*
bin	Executable programs
include	'C++' include files for development
lib	Libraries for development
config	Configuration utilities and template configuration files
fpga	FPGA firmware
stateTables	The Cycle Parameter tables
rootfs	The master root file system for the module controllers.
rootfs-[1234]	Copies of the master root file system for the individual module controllers. These are mounted as the root file system for the module controllers.
tmsExamples	Development example code
data	Data files such as test signals
html	HTML root for the TmsWeb program

tftpboot	The module controllers boot files master. These are copied into /tftpboot
doc	Documentation on the system.

The TMS TmsServer program's configuration file is in /etc/tmsServer.conf

7. System API's

There are four main API's used within the TMS. They are:

- **PupeApi:** This provides control and data access to the PUPE's individual, FPGA firmware implemented, Pick-Up channels. The interface implements a register level interface for control and a shared memory interface for data access. It also supports a DMA interface for fast data access. It is documented in the "PupeFpgaDesign" document.
- **Module Controller API (PuApi):** This provides software access to the individual Pick-Up processing engines. The API implements a simple RPC network API to allow control and data access to the individual Pick-Up channels. It also implements a system control, configuration and test API. It is documented in the "TMS API" document.
- **System Controller API (TmsApi):** This provides software access to the whole of the TMS system. The API implements a simple RPC network API to allow control and data access to the individual Pick-Up channels. It also implements a system control, configuration and test API. It is documented in the "System Controller Controller API" section in this document.
- **System Controller HTTP (Web):** This provides a basic HTTP protocol interface to the TMS system for viewing the status of the system and accessed raw data.

Each individual Pick-Up channel is allocated a Logical and Physical channel identifier. The Logical channel identifier is a number between 1 and 40 and defines the PU channel number on a particular ring. The Physical channel identifier consists of three parts: The Module Controller number (1 - 4), the Pupe Engine number (1 – 5) and the Pupe Channel number (1 – 3). The systems configuration defines a mapping between the logical and physical identifiers. This allows boards to be substituted while the system is running by simply moving the input ADC lines and reconfiguring the logical to physical Pick-Up channel number table.

A logical channel identifier of 0 is used to mean all channels on a ring. This use of 0 is also functional in the physical identifier components to define all Module Controllers, all PUPE Engines on a Module Controller and/or all channels of a PUPE Engine.

The Beam BOAP Object based RPC mechanism is used for the RPC. This provides a simple and efficient binary object based RPC mechanism with event capability.

7.1. System Controller API (TmsApi)

Generally users of the system are only concerned with the top level, System Controller API. This is the API that control and data gathering clients use to access the system. The System Controller API (TmsApi) is implemented using a simple, object orientated, RPC mechanism. Two main objects, the TmsControl and TmsProcess objects, provide the full API.

The TmsApi has been developed using the BOAP (BEAM Object Access Protocol). This provides a simple but powerful and efficient Object Orientated RPC mechanism. The TmsApi is written in a high level interface definition language (IDL). The **bidl** tool generates the client and server side 'C++' interface

and implementation files for the API. These are then provided as a set of 'C++' header files and a binary library file for the client applications to link to. The BOAP system employs a simple BOAP name server process that provides a translation between object names and the TCP/IP Address/Socket numbers that are used for object communications. The BOAP name server runs on the System Controller. More information on the BOAP system can be found in the libBeam documentation.

The TmsControl object is used for system configuration, testing and diagnostics. The TmsProcess object is used by normal clients for Proton Synchrotron (PS) Cycle information configuration and data access.

The API is documented in the TmsApi document. An overview of the main API objects and their basic functionality is given later in this document. There are some example client code in the tmsExamples directory of the source code and is also listed in the TmsApi document.

Each client application connects to one or both of these control objects through a TCP/IP network connection. The System Controller operates as a multi-threaded process and can communicate with multiple clients simultaneously.

The TMS system takes most of its system timing signals from digital timing lines connected to the TMS rack hardware. The only timing information that external software needs to supply is the next cycle number and cycle type information. This information consists of a 32bit unsigned number identifying the next Proton Synchrotron (PS) machine cycle and an ASCII string defining the beam type of the cycle. The CERN client software needs to provide this information by calling the setNextCycle() function at least 10ms before the next PS cycle is initiated.

A client application would generally use the TmsProcess object for its interface to the TMS system. It would use the getData() method to fetch the required data from the system. There is also an event based data interface implemented using the requestData() call and the TmsEvent event object.

Each of the TMS API calls return an error object. If there is an error, an appropriate error number will be given together with a string describing the error.

7.1.1. TMS Control Object (TmsControl)

This is responsible for overall control of the TMS and for configuring and getting statistics from the system. There follows an overview of the functions provided. Full details is provided in the TmsApi document.

Function	***Description***
init()	Initialises the system including resetting all of the PUPE engines firmware. The call will return an error object indicating success or an error condition as appropriate.
setProcessPriority(UInt32 priority)	Sets the priority of the process servicing this service
configure(ConfigInfo configInfo)	Configure the system for use. This includes mapping the individual physical Pick-Up channels to logical pick-up channels.
setControlInfo(CycleParam params)	Adds the state/phase control information for the given cycle type to the TMS systems database of cycle types. The parameters for the FPGA processing are passed. This includes the Phase and State table information. The call will return an error object indicating success or an error condition as appropriate.

Function	*Description*
delControlInfo (BString cycleType, UInt32 ring, UInt32 puChannel)	Deletes the control information for the cycle type, ring and puChannel number given. The call will return an error object indicating success or an error.
getControlInfo(BString cycleType, UInt32 ring, UInt32 puChannel, CycleParam& cycleParam)	Gets the Cycle parameter information for a given Cycle Type.
getControlList (BList< CycleParamItem > &itemList)	Gets the list of Cycle Parameters present in the system.
setNextCycle(UInt32 cycleNumber, String cycleType)	Sets the cycle number and cycle type for the next processing cycle. The call will return an error object indicating success or an error condition as appropriate. This should be called at least 10ms before the next CYCLE_START event.
test(ErrorList& errorList)	Performs a basic test of the system returning a list of errors. The call will return a list of error objects indicating error conditions that exist as appropriate. If no errors exist the call will return no error objects.
getStatus(NameValueList& status)	Returns the current status of the system. This information includes the number of Pick-Up's present and their individual status.
getStatistics(NameValueList& stats)	Returns a list of the statistic values as name/value pairs. The call will return an error object indicating success or an error condition as appropriate.
getPuChannel(int puChannel, PuChannel& puPhysChannel)	Returns the physical PU channel given a logical channel number. This uses the channel configuration table to look up the physical channel identifier. This function is provided so that the individual Pick-Up test functions can be accessed etc.
setTestMode(PuChannel puPhysChannel, UInt32 testOutput, UInt32 timingDisableMask)	The signal source for the digital test output connector. 0: None, 1: FrefLocal. The timingDisableMask bit mask defines which of the timing inputs should be disabled. If a timing input is disabled it can be still operated by software command.
setTimingSignals(PuChannel puPhysChannel, UInt32 timingSignals)	This function sets the given timing signals to the values as defined in the timingSignals bit array.
captureDiagnostics(PuChannel puPhysChannel, TestCaptureInfo captureInfo, Array<UInt64>& data)	This function will capture diagnostics test data. See the section on diagnostics capture for more details.
setTestData (PuChannel puPhysChannel, Int32 on, BArray< UInt32 > data)	This function will set a PU channel to sample data from test data memory rather than the ADC's. The data parameter points to an array of test data to use. See the section on "Using Test Data" for more information.
setPupeConfig (PuChannel puPhysChannel, PupeConfig	Sets special PUPE configuration for test purposes. The PupeConfig structure defines the settings.

Function	*Description*
pupeConfig)	

7.1.2. TMS Process Control Object (TmsProcess)

This object controls the TMS cycle processing and data gathering functions.

Function	*Description*
getCycleInfo(out UInt32 cycleNumber, out String cycleType)	Gets the current cycle number and type.
getCycleInformation(UInt32 cycleNumber, CycleInformation& cycleInformation)	This function gets detailed information on the given cycle number. This includes the timing of all CyclePeriods and the amount of data captured.
getCycleTypeInformation(String cycleType, CycleTypeInformation& cycleTypeInformation)	This function returns detailed information on the given cycle type. This includes the mask for buckets to bunches lookup.
getData(DataInfo dataInfo, Data& data)	This function returns a set of data from the data present in PUPE engines memory. The DataInfo object describes the data required. The call will return the required data along with an error object indicating success or an error condition as appropriate. The call will block until data is ready.
requestData(DataInfo dataInfo)	This adds a request for some data. The DataInfo object defines the data required. This request can be made at any time. The call will return immediately. The system will await the data from a subsequent processing cycle. When the data is available a “dataEvent” will be sent to the client. Note that it is not necessary to use requestData. The client can call getData() directly although this call will block until the data is actually ready.
addEventServer(String name)	This call adds an event server to call on events such as the “dataEvent” generated by the requestData() call as well as error events. The Client will use this to notify the TmsServer of its local TmsEvent object.

7.1.3. TMS Event Object

This event server object is created by client applications that are interested in getting asynchronous events from the TMS system. The Client should use the addEventServer() call in order to register the event server with the TMS system. The TmsEvent object provides the following local calls:

Function	*Description*
errorEvent(in UInt32 cycleNumber, in Error error)	This event function gets called on a system error. The errorEvent object contains and error number and string describing the error. The getStatus() call can be used to fetch further information.
cycleStartEvent(in UInt32 cycleNumber)	This event function gets called on the CYCLE_START event with the cycle number about to be processed

Function	*Description*
cycleStopEvent(in UInt32 cycleNumber)	This event function gets called on the CYCLE_END event with the cycle number completed
dataEvent(in DataInfo dataInfo)	This event function gets called when some requested data becomes available. The DataInfo object contains information on the data. The getData() call can be used to fetch the actual data.

7.2. TMS Data Client access

Most of the TMS activity will centre on the client applications accessing the data acquired by the TMS system. The system can support any reasonable number of client applications accessing the TMS data limited by the overall system bandwidth and the actual amount of data read. The system has storage for around 10GBytes of real-time data which equates to about 3 seconds of data capture. The client applications can access this data while it is available in the memory of the PUPE processing engines. There is thus around a 2 second window of opportunity in which to read the data acquired. The TMS is connected to the client systems by means of a single Gigabit Ethernet interface. The maximum client data rate is about 62 MBytes per second.

Client applications access the TMS data by using the **TmsProcess** API object. Each **TmsProcess** object, that is connected to the the TMS system, operates over a separate TCP/IP socket interface and has its own processing thread within the TmsServer process. In addition each **TmsControl** object is thread safe in that it is locked while a RPC is taking place. This allows easy use in multi-threaded applications.

An example of a client application reading some data is given in the example code file tmsDataClient1.cpp. The contents of this file is listed below:

```
/*****************************************************************************
 *      TmsDataClient.cpp    TMS API example code for a Data Client
 *                      T.Barnaby,     BEAM Ltd,     2007-02-07
 *****************************************************************************
 *
 *      This is a very basic example of using the TmsApi from a clients perspective.
 *      It is designed to give an overview of using the API.
 */
#include <iostream>
#include <stdio.h>
#include <TmsD.h>
#include <TmsC.h>

using namespace Tms;
using namespace std;


// Function to reads some data
BError tmsTest(TmsProcess& tmsProcess){
        BError                  err;
        DataInfo                dataInfo;
        Data                    data;
        UInt32                  cn = 0;
```

```
	BString			ct;

	// Find out the current cycle number and type
	if(err = tmsProcess.getCycleInfo(cn, ct)){
		return err.set(1, BString("Error: Getting Cycle Number: ") + err.getString());
	}

	printf("Getting data for cycles starting at cycle: %u\n", cn);

	for(; ; cn++){
		// Set data require and wait for data
		printf("GetData: Cycle Number: %u\n", cn);
		dataInfo.cycleNumber		= cn;
		dataInfo.channel	= 1;
		dataInfo.cyclePeriod	= CyclePeriodHarmonic0;
		dataInfo.startTime	= 0;
		dataInfo.orbitNumber		= 0;
		dataInfo.bunchNumber		= 0;
		dataInfo.function	= DataFunctionRaw;
		dataInfo.argument	= 0;
		dataInfo.numValues	= 1024;
		dataInfo.beyondPeriod		= 0;

		if(err = tmsProcess.getData(dataInfo, data)){
			return err.set(1, BString("Error: Getting Data: ") + err.getString());
		}
		printf("Data: NumValues: %d\n", data.numValues);
	}

	return err;
}

int main(int argc, char** argv){
	BError				err;
	TmsProcess		tmsProcess("//localhost/tmsProcess1");

	// Run a normal data gathering cycle as a normal client would.
	if(err = tmsTest(tmsProcess)){
		cerr << "Error: " << err.getString() << "\n";
		return 1;
	}

	return 0;
}
```

The TmsProcess object is used for communications with the TMS server. This is connected to the TMS using the connectService() call. The connectService call takes, as an argument, the host name of the TMS system and name of the BOAP object to connect to. The BOAP object name has the ring number

appended to it. This is encoded in a URL like format.

Once the TmsProcess object has been successively connected then the client can access the data using the getData() call. The getData call takes, as an argument, a DataInfo object that defines the data required. In this simple example the client application first uses the getCycleInfo call to determine the TMS's current cycle number and then attempts to read as many sets of the same data from the TMS.

The DataInfo object has the following fields:

cycleNumber	The PS Cycle number. This defines the TMS cycle number from which to get the data. If this number is in the future the getData call will block until the data is available. If the data for this cycle has all ready gone, then the "ErrorDataGone" error will be returned.
channel	The pick-up channel number. If this value is given as 0, then the TMS system will retrieve the data from all of the channels. Note that this could take significant time, especially for large values in numValues, and could result in an "ErrorDataGone" error.
cyclePeriod	The cycle period the data is from. Each processing cycle is split up into separate periods. These periods are: CyclePeriodAll – All of the processing cycle, CyclePeriodCalibration – The calibration period, CyclePeriodHarmonic0 – The period after injection to the first harmonic change, CyclePeriodHarmonic1 – the period after the first harmonic change to the next etc.
startTime	The start time in milli-seconds in the cycle period from which to fetch data, starting from 0.
orbitNumber	The starting orbit number, starting from 0. This is applied after the start time parameter.
bunchNumber	The Bunch number, starting from 1. If this is set to 0 the the data for all bunches is returned.
function	The data processing function to perform. This defines the type of data to be returned and any post processing to be performed. Currently there are three values for this: DataFunctionRaw: This specifies normal raw data as processed by the PUPE processing. DataFunctionMean0: This returns average values for all bunches with a 1ms resolution. The values from all bunches is averaged to a single data value each ms. DataFunctionMean1: This returns average values for the first bunch with a 1ms resolution. Note that the operation of DataFunctionMean0 and DataFunctionMean1 is performed in the PUPE FPGA and is configured by means of the State/Phase table parameters.
argument	The Argument to the data processing function. This is not currently used but is intended to provide a simple argument to the processing function.
numValues	The total number of data points to return
beyondPeriod	If set to 1 allows the reading of data beyond the period specified. Note that the number of bunches captured beyond the end of a cycle period may have changed.

The getData call returns the data in the Data object. This primarily consists of an array of 64 bit values.

The function also returns a BError object. This object defines if the function call was successful or not with an appropriate error number and error string. The possible errors are listed in the TmsSoftware document. The most likely errors are listed below:

ErrorCycleNumber	The Cycle Number and Type was not updated in-time for this cycle.
ErrorDataNotAvailable	The required data is not available. This means that there is no data for the given cycle number and/or period requested.
ErrorDataGone	The required data has already been overwritten by new data. This means the client was too slow in fetching the data or the TMS system was heavily loaded and could not supply the data before it had gone from the PUPE data memory.
ErrorDataFuture	The required data is to far into the future. This means that the cycle number requested is too far into the future.

The call returns a “Data” object. This has the following fields:

numValues	The total number of data samples in the dataValues array.
dataType	The type of data in the data block. Only the types DataTypeRaw and DataTypeMean are currently supported.
numBunches	The number of bunches of data present in dataValues.
numChannels	The number of channels of data present in dataValues.
dataValues	The data array.
errors	Individual errors for each channel within dataValues.

If the channel parameters has been set to 0 to capture data from all of the channels, and an error occurs for one or more channels, the “getData” call will continue to read data from as many channels as it can. In this case the call will return the first error that occurred as the function return value. The “errors” array in the “Data” object will contain the individual channel errors. The Data from any channel that had an error will be set to 0.

The getData call will check to see if the data for the cycle number requested is still present in the PUPE memory. The PUPE memory has enough storage for about 3 seconds worth of data (3 processing cycles). If the data has gone the call will return the error "ErrorDataGone". If the system has not processed the requested cycle, but will do so within 256 seconds, the call will block awaiting the data.

If the channel number is given as 0 the call will interrogate each of the Pick-Up channels and return the combined data from all of them. Note that this could take significant time and may not be possible if the parameter numValues is large. Within the Data structure returned there is an array of error values, one per channel. If an error occurs on any set of the channels the call will return the first error that occurred and the complete list of errors in the errors array. The actual data will be returned for all channels that did not have an error. Those channels that had an error will have data values of 0 returned.

If the bunch number is given as 0, then the system will return the data for all of the bunches.

The data will be returned in the following order, where B - Bunch, C - Channel:
[C1.B1, C1.B2, C1.B3, C1.B4], [C1.B1, C1.B2, C1.B3, C1.B4], ... [C2.B1, C2.B2, C2.B3, C2.B4], [C2.B1, C2.B2, C2.B3, C2.B4], ...
That is the data is ordered by bunch, then sample, then channel.

See the TMS Software documentation manual for more details of this functions operation.

7.3. The Raw Data

When the DataInfo's function parameters is set to “DataFunctionRaw” the getData call will return the raw pick-up data. The raw pick-up data is the individual Sigma, DeltaX and DeltaY samples integrated over one bunches period.

Each individual data item has the following members:

sigma	A 16 bit signed value giving the value of Sigma.
deltaX	A 16 bit signed value giving the value of DeltaX.
deltaY	A 16 bit signed value giving the value of DeltaY.
time	A 16 bit unsigned value giving the time in milliseconds from CYCLE_START that the sample was captured.

7.4. The Mean Data

When the DataInfo's function parameters is set to “DataFunctionMean” or “DataFunctionMeanAll” the getData call will return the averaged pick-up data. The averaged pick-up data is the sum of individual Sigma, DeltaX and DeltaY samples over each millisecond period divided by the number of samples.

The “DataFunctionMean” function returns the average data for any particular bunch or the whole set of bunches if the DataInfo parameter bunchNumber is set to 0.

The “DataFunctionMeanAll” function returns the average of all bunches. That is every bunch's values are averaged together with every other bunch to give a single set of values.

Each individual data item has the following members:

sigma	A 16 bit signed value giving the average value of Sigma.
deltaX	A 16 bit signed value giving the average value of DeltaX divided by 256.
deltaY	A 16 bit signed value giving the average value of DeltaY divided by 256.
time	A 16 bit unsigned value giving the time in milliseconds from CYCLE_START that the sample was captured.

8. TMS Testing

The TMS system provides a number of features to support testing the system. The main features include:

- The ability to load test Fref,Sigma,DeltaX and DeltaY data into the SDRAM of a PUPE board and use this as a source for the FREF timing signal and the three ADC input signals.
- The ability to generate the setNextCycle() calls automatically on the CYCLE_STOP event.
- The ability to generate some or all of the TMS timing signals in software to simulate the PS digital timing inputs.
- A diagnostics capture function so that internal information from the PUPE's FPGA can be captured. This is useful to check the PLL's operation and for raw ADC data capture.
- The tmsControl and tmsControlGui test programs provide the ability to access the diagnostics

functions from a command line or GUI based application.

- The tmsTestData program uses the TMS testing features to test accessing data from the system.

The TmsTesting document provides more information on using the testing functions.

8.1. Using Test Data

The **setTestData** API call allows the user to configure an individual pick-up channel to run with test data from the PUPE's TEST SDRAM bank. The call takes an array of 32bit data values to use. Each 32bit data item has the following structure:

Bits	31:22	21:12	11:1	0
Function	ΔX	ΔY	Σ	FREF

There has to be an even number of samples sent to the system. Only one set of test data can be used in an individual PUPE, however each of the 3 PUPE channels can be independently set to use the test data. Note that the FREF signal will be used as the FREF timing signal for any channel configured to use the test data source. Care should be used to set the PllPhaseDelay and/or FREF phase in the test data as appropriate so that FREF and Sigma/DelyaX/DeltaY sources are of the correct phase.

8.2. Software Generation of setNextCycle() calls

The TmsServer programs configuration file, /etc/tmsServer.conf, has a parameter named: **SimulateNextCycle.** If this parameter is set to 1 the TmsServer will automatically call the setNextCycle() function on each CYCLE_STOP event with an incrementing cycle number and the same cycle type.

8.3. Software timing

A TmsPuServer program can be set to drive the Module global TMS timing signals from software. This can be done by modifying the individual TmsPuServers configuration file parameter: **SimulateTiming**, or by using the TMS API's setSimulation or setPupeConfig calls. Normally this is done on a master PUPE in a rack all of the other PUPE engines in the rack will use the timing signals as supplied from the timing bus.

The software timing system is able to generate the CYCLE_START, CYCLE_STOP, INJECTION and one H-CHANGE event in a 1.2 second cycle.

In the setPupeConfig call the **SimulateTiming** parameter is a bit mask which defines which of the timing signals should be simulated in software. This can also be set or cleared using the ***setPupeConfig*** API call.

It is also possible for an external program to manually operate the timing signals by making use of the API's *setTestMode* and *setTimingSignals* calls.

8.4. Diagnostics Capture

The TMS API and PUPE FPGA firmware, provides the ability to capture diagnostics information from the FPGA. The diagnostics function has flexible clocking and triggering facilities and the ability to capture 64bits of data from 4 separate sources. The API's captureDiagnostics function provides the ability.

It takes an input parameter structure that defines the data capture required. This structure has the following parameters:

Name	***Description***

source	The source data one of 64bits (0 - 3)
clock	The Clock source (0 – 17). See below for settings.
startTime	The start time in ms from CYCLE_START before trigger is activated. Note current FPGA implementation only allows one of startTime or postTriggerDelay to be used.
postTriggerDelay	The delay, in clock cycles, after the trigger before capture starts. Note current FPGA implementation only allows one of startTime or postTriggerDelay to be used.
triggerMask	The Trigger bit mask. This is the bit mask of the 8 timing signals. See below for trigger bit settings.
triggerAnd	The Trigger function is an AND rather than an OR
triggerStore	Store the trigger data in the lower 8 data bits of the 64bit result.
triggerSourceData	Use lower 32bits of data as trigger source rather than timing signals

Clock Sources

Number	***Description***
ClkAdcDiv_1	ADC Clock
ClkAdcDiv_2	ADC Clock divided by 2
ClkAdcDiv_5	ADC Clock divided by 5
ClkAdcDiv_10	ADC Clock divided by 10
ClkAdcDiv_20	ADC Clock divided by 20
ClkAdcDiv_50	ADC Clock divided by 50
ClkAdcDiv_100	ADC Clock divided by 100
ClkAdcDiv_200	ADC Clock divided by 200
ClkAdcDiv_500	ADC Clock divided by 500
ClkAdcDiv_1000	ADC Clock divided by 1000
ClkAdcDiv_2000	ADC Clock divided by 2000
ClkAdcDiv_5000	ADC Clock divided by 5000
ClkAdcDiv_10000	ADC Clock divided by 10000
ClkAdcDiv_20000	ADC Clock divided by 20000
ClkAdcDiv_50000	ADC Clock divided by 50000
ClkAdcDiv_100000	ADC Clock divided by 100000
ClkFref	FREF
ClkMs	1ms timer

Timing Bit Mask

Bit	*Description*
0	10MHz System Clock
1	CYCLE_START
2	CYCLE_STOP
3	CAL_START
4	CAL_STOP
5	INJECTION
6	HCHANGE
7	FREF

See the API documentation for further details on the parameter settings. The source data bits are subject to change as the diagnostics are developed as required.

The diagnostics data consists of the following signals:

Source	*Bits*	*Description*
0	63:0	Undefined as yet
1	63:0	Undefined as yet
2	63:0	Undefined as yet
3	63:0	Undefined as yet

9. Multiple System Controllers

Two system controllers are provided, one as a backup or backup/development server. There are many ways to configure the system with two servers, the following briefly describes a possible configuration.

- Two TmsServers: 192.168.100.1 and 192.168.100.2
- Both systems can be setup identically apart from:
  - DHCP disabled on second server
  - TmsServer disabled on second server
  - tmsServer.conf and tmsPuServer.conf files set to use second server.

With this setup the second server can be brought on-line by simply enabling the DHCP service and power cycling the TMS modules.

The Second Server could be also be configured for development. In this case it would be configured to manage the spare module with 3 PUPE boards. The second server would supply DHCP information for spare module controller and the server's tmsServer.conf and tmsPuServer.conf files would be set to use second server (“TmsServer:” parameter).

10. FPGA Bit Files

The system uses the FPGA bit file, named tms-fpga.bit. This is stored, by default, in the /usr/tms/fpga

directory. This bit file is loaded into each PUPE on initialisation by the TmsPuServer program. Normally this file is provided in the tms-fpga RPM software package.

The FPGA's are loaded with this firmware on boot and whenever the init() TMS API function is called.

11. Further Software Documentation

There are further TMS Software documents available on the CERN TMS Support website at: https://portal.beam.ltd.uk/support/cern/.

The lower levels of the code are documented using the the DOxygen tool to create class and function level documentation.

12. Error Handling

There are a number of possible sources for errors to occur while the TMS system is running and a number of ways of dealing with the errors. All errors that can occur have an error number and an error string. Errors are reported to the client applications by means of a returned error object and/or the TmsEvent system. The Following table lists the main system errors that can occur. See the individual API calls for specific information on errors returned.

Error	*Description*
ErrorOk	No Error. This is the status returned when the command completed with no errors.
ErrorMisc	A miscellaneous unclassified error occurred.
ErrorWarning	A warning message. No actual error occurred.
ErrorInit	An error occurred during initialisation of the system.
ErrorConfig	There is an error in the system configuration files.
ErrorParam	There was an error in one of the parameters passing in an API call.
ErrorNotImplemented	This function has not been implemented.
ErrorComms	A communication error occurred.
ErrorCommsTimeout	A communications time out occurred.
ErrorMC	A Module Controller has an error
ErrorFpga	There is an error with a PUPE FPGA board.
ErrorStateTable	An error event occurred due to an incorrect FPGA State table transition.
ErrorCycleNumber	The Cycle Number and Type was not updated in-time for this cycle.
ErrorDataNotAvailable	The required data is not available. This means that there is no data for the given cycle number and/or period requested.
ErrorDataGone	The required data has already been overwritten by new data. This means the client was too slow in fetching the data of the TMS system was heavily loaded and could not supply the data before it had gone from the PUPE data memory.
ErrorDataFuture	The required data is to far into the future. This means that the cycle number requested is too far into the future.

13. Software Distribution and Updates

The complete software and documentation for the TMS system is available on DVD and on the Beam CERN support web site. This includes a full installation package together with individual packages in RPM format. Complete source code is also available in raw form and through the SVN version control system.

All of the software is installed on the System Controllers and is packaged as Linux RPM packages. This enables easy and controlled software updates to the system. It is possible to update the second, spare System Controller and test the system while the primary System Controller is in use. It is then be possible to restart the system using the second, spare controller as the master controller.

For more details on this see the TMS Maintenance manual.