CERN Trajectory Measurement System

Software Design

Project	CERN-TMS
Date	2006-11-17
Reference	Cern-tms/softwareDesign
Version	1.0
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.
- Alpha Data's TMS Tender "pre-design-1.5".
- Emailed questions answered by Jeroen Belleman of CERN.
- Visit to CERN on 2006-06-20
- TMS design documents systemDesign, pupeFpgaDesign, pupeBoardDesign.
- TMS Development and Support website at: https://portal.beam.ltd.uk/support/cern/.

2. Introduction

This Software Design document concerns the high level design of the software for the CERN Trajectory Measurement System (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, data access and test functions for the system.

To gain an understanding of the overall systems design please refer to the systemDesign document

3. System Software 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 will be based on the Linux operating system. This will provide a reliable and flexible system that can be easily maintained locally and remotely. All of the software will Open Source and thus all source code will be available.

All communications with external systems will be through the system controller (SC) which will support a simple API to control and gather data from the system. The system controller will interrogate 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 software will be developed on the GNU/Linux operating system using the Open Source GNU tool-set. The software will be predominantly written in the 'C++' language.

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).

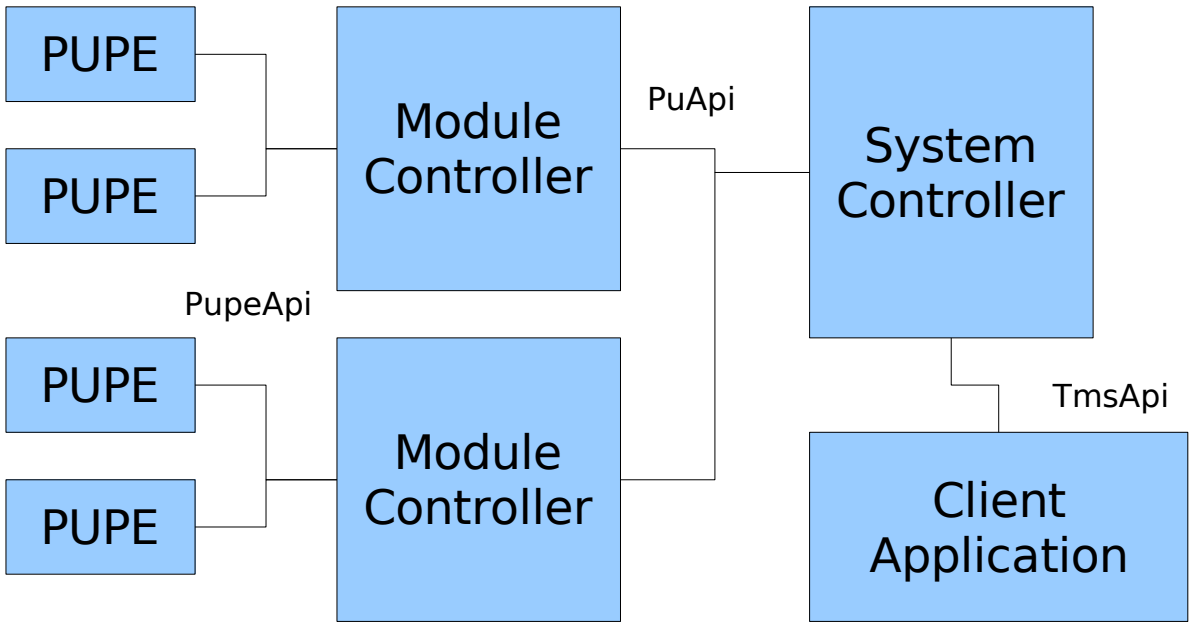

Illustration 1: TMS Main System Modules

The TMS has a private 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 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.

The PUPE is 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 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 will have an Intel x86 based CPU, some boot FLASH memory, 1 Gigabyte of RAM, a cPCI bus interface and a Gigabit Ethernet port.

The module controller will boot from the main system controller (SC) over the Ethernet interface and will run a small Linux based operating system. It will be responsible for booting and managing the 5 PU processing engines (15 Proton Synchrotron PU's) on its bus. Communications between the system

controller and the individual PU processing engines will also be handled.

The Module Controller implements a simple network based API for control and access to the individual PUPE channels.

3.3. System Controller (SC)

The system controller will be a standard Intel Xeon based computer system. It is housed in a separate 2U or 4U 19" rack enclosure. The system controller will have 2 Gigabyte's of memory and dual disk drives in a RAID configuration for disk redundancy. These disks will contain all of the TMS's software, FPGA firmware and configuration information. The controller will have dual Gigabit Ethernet interfaces, one connected to the Gigabit switch that communicates with the processing module's controllers and one connected to the sites LAN for remote access to the system.

The system controller will not have a monitor, keyboard or mouse. All system configuration and maintenance will be carried out over the Ethernet network. The system controller will run the Linux operating system.

Two identical system controllers will be provided for system redundancy.

As well as providing a control and data interface to the Trajectory Measurement System, the software on the system controller will implement system boot, system configuration, system test and fault diagnostics functions. This will be made available to operators via a web based interface as well as through a command line API.

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. However, it will also be possible for CERN to implement these applications on the TMS's system controllers if desired. These 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 will be developed within the Linux operating system environment and will use the Linux operating system as its base system layer. The base Linux distribution we will use will probably be Fedora Core 6, using the 2.6.x Linux kernel. The TMS individual systems, the MC and SC, will only have a limited installation of this operating system especially the Module Controllers.

The main software development tools will be the GNU tool-set and the development language will be 'C++'. These tools will be installed on the System Controllers but can be used on a separate Fedora Core

6 system. The SVN version control system will be used for version control.

All software will be supplied in source code as well as binary forms.

5. System API's

There are three 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 will also possibly support a DMA for 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 "Module Controller API" section in this 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.

Each individual Pick-Up channel has two number's associated with it. One is the physical position of the Pick_Up within a single TMS module. This number is in the range 0-17. The second number is the logical Pick-Up number within the TMS. This number is in the range 0-999. A configuration mapping table links the logical Pick-Up number to the physical module number and individual Pick-Up within the module. 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.

We are intending to use the Beam BOAP Object based RPC mechanism for the RPC. We have used this for a number of projects. It provides a simple and efficient object based RPC mechanism with event capability.

6. Module Controller API (PuApi)

The Module Controller API (PuApi) will be implemented using a simple, object orientated, RPC mechanism. A number of objects will be created each implementing a portion of the overall API. The main API objects and their basic functionality are listed below, this API will be developed during the software development phase of the project. The API has been designed so that the PUPE engines could be accessed directly through their on-board Gigabit Ethernet interfaces in the future.

The data definition objects have yet to be fully defined as we are awaiting details on the type of data

requests required so that we can design and implement them in an efficient manner. It is expected that the DataInfo object will allow remote pre-processing of the data to be performed using user defined functions.

6.1. Pick-Up Control Object (PuControl)

This is responsible for overall control of the Module controller and for configuring and getting statistics from the system.

Function	*Description*
init()	Initialises the system including loading all of the PUPE engines firmware. The call will return an error object indicating success or an error condition as appropriate.
test(ErrorList& errorList)	Performs a basic test of the system returning a list of errors. The call will return an error object indicating success or an error condition as appropriate.
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.

6.2. Pick-Up Process Object (PuProcess)

This object provides process control on an individual Pick-Up on one of the PUPE's under the Module Controllers control. The functions are passed a puChannel number. This is used when accessing the PuChannels through the MC.

Function	*Description*
cycleStart(int puChannel, PuCycleParam params)	Start off a processing cycle. The Pick-Up channel's physical number is passed (0 through 17 on a module controller) and parameters for the processing cycle are passed. The call will return an error object indicating success or an error condition as appropriate.
cycleWaitForEnd(int puChannel)	Wait for the completion of a processing run. The call will return an error object indicating success or an error condition as

Function	*Description*
	appropriate.
getStatus(int puChannel, PuStatus& status)	Returns the current status of the Pick-Up engine.
getData(int puChannel, DataInfo dataInfo, Data& data)	This function returns a set of data from the data present in the Pick-Up processing engine. The function is given a DataInfo object describing the data required. The call will return the required data along with an error object indicating success or an error condition as appropriate.
setTestMode(int puChannel, UInt signal, Uin32 timingDisableMask)	The signal source for the digital test output connector. 0 – None, 1 - FrefDelayed, 2 – PllFRef, 3 – PllFRefHarm. 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(int puChannel, UInt32 timingSignals)	This function sets the given timing signals to the values as defined in the timingSignals bit array.
captureTestData(int puChannel, UInt32 sampleClock, UInt32 delayMs, Bool triggerAnd, UInt32 triggerBits, void*data)	Captures diagnostics data and returns the data. The sampleClock parameter defines the capture rate and sample clock source. The sample clock sources are defined below. The delayMs defines the time to wait, in milliseconds, after CYCLE_START before looking for a trigger event. The triggerAnd parameter defines whether to use an OR(0) or AND(1) function between the triggerBits. The triggerBits parameter defines the trigger pattern to start capture. The actual trigger pattern bits have yet to be defined but they will include the systems timming signal inputs together with some important internal signals. The test data consists of 128bit values. Each 128bit value has the following format: [PllFrequency 24][PllTheta 24][PhaseTableSignals 8][TimingSignals 8][Spare 16][DeltaY 16][DeltaX 16][Sigma 16] Note the format and contents of this data are subject to change.

Sample Clock Parameter		
Bits15:8	**Sample Clock**	**Frequency**
0	ADC CLK	125MHz
1	ADC CLK /2	62.5MHz
2	ADC CLK /5	25MHz
3	ADC CLK /10	12.5MHz
4	ADC CLK /20	6.25MHz
5	ADC CLK /50	2.5MHz
6	ADC CLK /100	1.25MHz
7	ADC CLK /200	625kHz
8	ADC CLK /500	250kHz
9	ADC CLK /1000	125kHz
10	ADC CLK /2000	62.5kHz
11	ADC CLK /5000	25kHz
12	ADC CLK /10000	12.5kHz
13	ADC CLK /20000	6.25kHz
14	ADC CLK /50000	2.5kHz
15	ADC CLK /100000	1.25kHz
16	SYSTEM_CLOCK	10MHz
17	C-CLOCK	1kHz
18	F_REF	200-500kHz
19	F_REF_DELAYED	200-500kHz

6.3. The PuCycleParam Object

The PuCycleParam object contains information on the next processing cycle. It has the following values:

Parameter	*Description*
UInt32 cycleNumber	The Cycle number.
UInt32 cycleType	The type of cycle
UInt32 frefPhaseDelay	The phase delay parameter for the Fref timing signal. This is set based on the position of the Pick-Up in the PS ring.
UInt32 pllInitialFrequency	This defines the initial PLL's frequency. This is loaded on START_CYCLE and after the delay given in pllInitialFrequencyDelay.
UInt32 pllInitialFrequencyDelay	This defines the delay in milliseconds from START_CYCLE when the pllInitialFrequency is loaded.
UInt32 pllGain	The gain of the PLL feedback system
UInt32 pllDdsMinimum	PLL DDS minimum frequency
UInt32 pllDdsMaximum	PLL DDS maximum frequency
UInt32 stateTable[16]	The array of State Table entries for the processing run
UInt8 phaseTable[16][512]	The array of Phase Table entries for the processing run
UInt32 numBunches[16]	The number of bunches captured, per orbit, within the given phase table period

6.4. The PuStatus Object

The PuStatus object returns status information on the Pick-Up processing engine. It has the following values:

Parameter	*Description*
Bool running	The Pick-Up is currently running.
Error error	The Pick-Up's current error status.

6.5. The DataInfo Object

The DataInfo object defines the data to be returned from the Pick-Up processing engine. This is just an idea at the moment. A set of data values can be captured. There are two ways of defining the period for data collection, one is to collect the data gievn a time in milliseconds and an orbit number the other is from a given processing cycles period. The cycle periods are those periods defined by the systems timing signals. These are: 1 – CalibrationPeriod, 2 - Harmonic0Period, 3 – Harmonic1Period, 4 –

Harmonic2Period ...

The requests asks for a certain number of data values to be returned. If a harmonic change occurs before this number of values is reached a smaller number of data values will actually be returned.

It has the following parameters:

Parameter	*Description*
UInt32 cycleNumber	The Cycle number.
UInt32 channel	The Pick-Up channel number. Logical or Physical depending on API layer.
UInt32 cyclePeriod	This defines what type of data to return. 0 – Use the startTime and orbitNumber defined in the following fields, 1 – CalibrationPeriod, 2 - Harmonic0Period, 3 – Harmonic1Period, 4 – Harmonic2Period ...
UInt32 startTime	The start time in milliseconds
UInt32 orbitNumber	The Orbit number defining the starting time for the data past the startTime. (Do we need time in ms as well ??)
UInt32 bunchNumber	The Bunch number for which to return data. (Could be a bunch mask ?)
UInt32 function	The data function to use. (0 – Raw Sigma/DeltaX/DeltaY, 1 – Lowpass Filtered data (1 sample per ms) Sigma/DeltaX/DeltaY, 2 – UserFunction)
UInt32 argument	Argument for data function. Unused in current system. Could be used for filter parameters etc.
UInt32 numValues	The total number of values to return

6.6. The Data object

The Data object describes the data returned from the Pick-Up processing engine. This is just an idea at the moment. It has the following parameters:

Parameter	*Description*
UInt32 numValues	The total number of values
UInt32 dataType	The type of data in the data block. 0 - [Spare,DeltaY,DeltaX,Sigma]

Parameter	*Description*
DataValue data[]	Array of data values (DataValue has the elements: [Spare,DeltaY,DeltaX,Sigma])

7. System Controller API (TmsApi)

The System Controller API (TmsApi) will be implemented using a simple, object orientated, RPC mechanism. A number of objects will be created each implementing a portion of the overall API. The main API objects and their basic functionality is listed below, this API will be developed during the software development phase of the project.

The data definition objects have yet to be fully defined as we are awaiting details on the type of data requests required so that we can design and implement them in an efficient manner. It is expected that the DataInfo object will allow remote pre-processing of the data to be performed using user defined functions.

7.1. TMS Control Object (TmsControl)

This is responsible for overall control of the TMS and for configuring and getting statistics from the system.

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.
configure(ConfigInfo configInfo)	Configure the system for use. This includes mapping the individual physical Pick-Up channels to logical pick-up channels.
test(StrList& errorList)	Performs a basic test of the system returning a list of errors. The call will return an error object indicating success or an error condition as appropriate.
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.

Function	*Description*
getPuProcess(int puChannel, PuProcess& puProcess, int& puPhysChannel);	Returns a reference to the Pick Up channels PuProcess object for the given logically numbered Pick-Up. This can be used so that the individual Pick-Ups test functions can be accessed etc.
errorEvent(Error errorEvent)	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.

7.2. TMS Process Control Object (TmsProcess)

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

Function	*Description*
setControlInfo(CycleParam params)	Sets the control information for the cycle number given and subsequent cycles. The parameters for the processing cycle 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.
setNextCycleNumber(UInt32 cycleNumber)	Sets the cycle number 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 100ms before the next CYCLE_START event.
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. If the data is present in cache the data will be available immediately, if not the system will await the data from a subsequent processing cycle. When the data is available a “data” event will be sent to the client. Not 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.
getData(DataInfo dataInfo, Data& data)	This function returns a set of data from the data present in the data cache or directly from the Pick-Up processing engines. 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

Function	*Description*
	is ready.
dataEvent(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.

7.3. The ConfigInfo Object

This object is used to define the configuration of the system.

Parameter	*Description*
PuReference puReferences[256]	The logical to physical Pick-Up table. Each PuReference includes a Module Controller identifier (Possibly IP address) and a Physical Pick-Up number.

7.4. The CycleParam Object

The CycleParam object contains information on the next processing cycle. It has the following values:

Parameter	*Description*
UInt32 cycleNumber	The Cycle number.
UInt32 frefPhaseDelay[128]	The phase delay parameters for the Fref timing signal for each of the Pick-Up channels. This is set based on the position of the Pick-Up's in the PS ring.
UInt32 pllInitialFrequency	This defines the initial PLL's frequency. This is loaded on START_CYCLE and after the delay given in pllInitialFrequencyDelay.
UInt32 pllInitialFrequencyDelay	This defines the delay in milliseconds from START_CYCLE when the pllInitialFrequency is loaded.
UInt32 pllGain	The gain of the PLL feedback system
UInt32 pllDdsMinimum	PLL DDS minimum frequency
UInt32 pllDdsMaximum	PLL DDS maximum frequency
UInt32 stateTable[16];	The array of state table entries for the processing run
UInt8 phaseTable[16][512]	The array of phase table entries for the processing run

Parameter	*Description*
Uint32 numBunches[16]	The number of bunches captured, per orbit, within the given phase table period

8. Module Controller Software

The Module Controller's will boot and get all of their configuration information from the System Controller. They will use the industry standard DHCP, TFTP, NFS and NTP protocols to achieve this. The system will boot the Linux 2.6.x kernel and run using a minimal Linux system based on the busybox application. They will use an initial RAM disk for the root file system during boot and then use an NFS supplied file system mounted from the System Controller.

As well as the busybox system application the system will run the **tmsPuControl** application. The **tmsPuControl** application implements the TmsPuApi for each of the Pick-Up engines under its control and is run as a "real-time" process.

The **tmsPuControl** applications is responsible for booting and managing the individual PUPE's and will communicate with them using the PupeApi.

9. System Controller Software

The System Controller will boot from its internal hard disks. It will run a reduced version of Linux Fedora Core 6. It will provide the following standard networking services to the Module Controllers:

- DHCP Server: For local network configuration.
- TFTP Server: For booting the Module Controllers kernel and initial RAM disk.
- NFS Server: Provides file system for Module Controllers.
- NTP Server: Provides date and time functions for the Module Controllers.

The system will support the IPMI serial over LAN control interface for managing low level BIOS access. However, the server will also have a VGA monitor port and keyboard port that can also be used for low level BIOS access. Either of these methods can be used if a complete, "bare metal", software installation is required.

The system will run a number of standard system processes and the **tmsControl** process. The **tmsControl** process will run as a "real-time" process and be responsible for implementing the TMS System Controller API (tmsApi). The **tmsControl** process will also implement a simple HTTP web-server interface to allow the current status and configuration of the system to be viewed in a conventional web-browser.

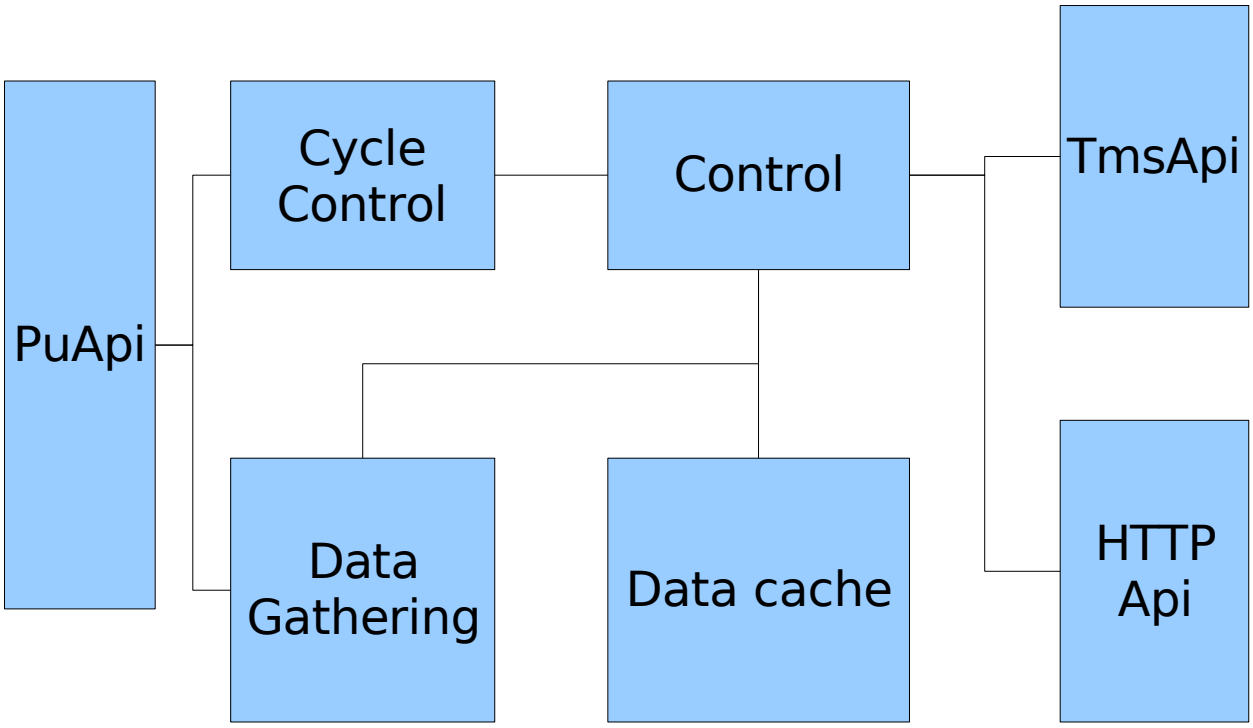

Illustration 2: TmsControl Block Diagram

The TmsControl application implements the main system control and data gathering functionality. It implements the TmsApi which client applications use to control the system.

10. Software Documentation

The software will be documented at a high level in conventional PDF format documentation files. The lower levels of the code will use the DOxygen tool to create class and function level documentation.

11. Software Distribution

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

12. Software Updates

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