ALPHA DATA
BEAM

# CERN Trajectory Measurement System

# Software Development Manual

| | |
|---|---|
| **Project** | CERN-TMS |
| **Date** | 2013-02-01 |
| **Reference** | Cern-tms/TmsSoftwareDevelopment |
| **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 document provides some additional information to assist in developing client software to access the TMS system. See the **TmsSoftware** manual for more details on the overall TMS software and TMS API documentation.

## 3. Overview

The TMS system provides a simple, network based, remote procedure call (RPC) mechanism for control of and data access from the TMS. This RPC is implemented using the BEAM BOAP (Beam Object Access Protocol). This is a simple and efficient object orientated RPC mechanism suited to this task.



Web: www.alpha-data.com, www.beam.ltd.uk
Email: terry.barnaby@beam.ltd.uk

# 4. Developing on the System Controller

All of the TMS software is installed in the /usr/tms directory. Included in this directory structure are the 'C++' header files and library binary files for the libTms and libBeam libraries. These are situated in the /usr/tms/include and /usr/tms/lib directories.

The libBeam library provides basic, low level, objects for developing applications. This includes objects for string, network and thread handling as well as the core BOAP functionality.

The libTms library provides the TMS API access library together with some helper definitions and functions for TMS access as well as CycleParameter table generation and management classes.

The libBDebug.a library provides debug utilities including crash backtrace system.

The libadmxrc2.so Alpha Data ADMXRC interface library. Uses the admxrc2 Linux kernel driver to communicate with the PUPE boards.

There are some example TMS client applications in the /usr/tms/tmsExamples directory.

If the System Controller has had the development system installed then it is possible to build client applications on the system. Use the examples in the tmsExamples directory as a template for this.

# 5. Developing on external systems

To develop TMS client software on external systems the libBeam and libTms libraries need to be ported to the external system. Either just the TMS client library source or the full TMS source code is available to be installed onto a system to aid this. The TMS API is written in the 'C++' language and is designed to be built using the GNU compiler tool set.

# 6. Controlling the TMS system

The TMS system is controlled through the **TmsAPI's TmsControl** object. This provides a number of functions that can be used to configure and control the system. This interface is used for the important **setNextCycle**() call as well as diagnostics functions.

Each **TmsControl** 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.

There can be multiple TmsServers for separate rings operating. Each of these are allocated a ring number from 1 to 4. The control objects name, for these servers, has the ring number appended so a client can talk to one of the rings control processes.

## *6.1. Using the setNextCycle call*

The **setNextCycle** call sets the cycle number and type for the next processing cycle. It is a crucial function that supplies the TMS system with information on the next processing cycle. The timing of the **setNextCycle** call is crucial to the operation of the system. The call should be made in the period after the previous cycles CYCLE_START event and 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 TMS plenty of time to load the FPGA's with the new state/phase tables.

The **setNextCycle** call is provided with cycle number and cycle type information. 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.

A simple example of some code using the **setNextCycle** call is given in the **tmsControlClient2.cpp** example source file. This is printed below:

```
/*****************************************************************************
 *      TmsControlClient2.cpp       TMS API example code
 *                  T.Barnaby,     BEAM Ltd,    2007-02-07
 *****************************************************************************
 *
 *      This is a very basic example of using the TmsApi to set the
 *      TMS's cycleNumber and cycleType.
 *      It is designed to give an overview of using the API.
 */
#include <iostream>
#include <stdio.h>
#include <unistd.h>
#include <TmsD.h>
#include <TmsC.h>

using namespace Tms;
using namespace std;

// Loop sening next cycle information
BError tmsControlLoop(TmsControl& tmsControl){
        BError                  err;
        UInt32                  cn = 0;
        BString                 ct = "Beam3";

        while(1){
                // Wait for next cycle information
                usleep(1200000);

                // Set next cycle information
                cn = cn + 1;
                ct = "Beam3";

                // Send the next cycle information to the TMS server
                if(err = tmsControl.setNextCycle(cn, ct)){
                        cerr << "Error: " << err.getString() << "\n";
                }
        }

        return err;
}

int main(int argc, char** argv){
        BError                  err;
```

```
        TmsControl          tmsControl;
        BString             hostName = "localhost";

        // Connect to the Control service
        if(err = tmsControl.connectService(BString("//") + hostName + "/tmsControl1")){
                cerr << "Error: " << err.getString() << "\n";
                return 1;
        }

        // Set the network priority high
        if(err = tmsControl.setPriority(BSocket::PriorityHigh)){
                cerr << "Error: " << err.getString() << "\n";
                return 1;
        }

        // Set the TmsServer thread priority high
        if(err = tmsControl.setProcessPriority(PriorityHigh)){
                cerr << "Error: " << err.getString() << "\n";
                return 1;
        }

        if(err = tmsControlLoop(tmsControl)){
                cerr << "Error: " << err.getString() << "\n";
                return 1;
        }

        return 0;
}
```

The TmsControl 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 which includes the ring number. This is encoded in a URL like format.

Once the TmsControl object has been successively connected then the system increases the priority of the Network Connection. On systems that support this, such as Linux, this increases the priority of packets sent over this TCP/IP link over other network packets passing between the systems. Currently this only increases the priority at the packet queues in the two communicating systems, but it can set the TCP/IP TOS QOS bits so that intervening network switches can honour the priority.

The setProcessPriority call is now used to increase the priority of the TMS internal thread handling this connection. This effectively increases the priority of the setNextCycle handling within the TMS server over normal data access functions.

Once the TmsControl TCP/IP link has been fully configured the setNextCycle call is called continually at the appropriate time. We would expect the call to be synchronised to the CERN PS system by means of current CERN control protocols. It would also be possible to set up the client application to receive events from the TMS system and to synchronise the call with the CYCLE_STOP event. There is a simple example, tmsControlClient3.cpp, that performs this.

The setNextCycle 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 error is the ErrorCycleNumber - "The Cycle Number and Type was not updated in-time for this cycle." This indicates that the setNextCycle function did not complete in time for the START_CYCLE it referred to.

### *6.2. Control and Diagnostics*

Most of the other TmsControl API functions are concerned with diagnostics and control. It is expected that these functions would only be used by engineers responsible for control and management of the system.

## 7. 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 including the ring number. 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 getData call returns the data in the Data object. This primarily consists of an array of 64 bit values containing the Time, Sigma, DeltaX and DeltaY components.

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 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. |

See the **TmsSoftware** manual for more details.

## *7.1. Data Access Latency and Speed*

When a client fetches data from the TMS system, it has to pass through the various system layers. In the current system implementation, a set of data cannot be streamed through these layers, it must be copied as a complete block of data. This results in some latency delays in the system. The latency delays following a getData call are:

- The TmsPuServer fetches data from a PUPE's memory. The cPCI bus data rate is around 100MBytes/sec.
- The TmsServer fetches the data from the TmsPuServer. The Internal Gigabit Ethernet transfer speed after overheads is about 62MBytes/sec.
- The clients fetches the data from the TmsServer. The External Gigabit Ethernet transfer speed after overheads is about 62MBytes/sec.

The actual latency is dependent on the size of data required. Although the amount of latency is not a real issue in the system, its effect on overall bandwidth could be. However, use can be made of the fact that the system is a multi-threaded SMP system. This allows multiple clients or single multi-threaded clients to access the data simultaneously and eliminate the bandwidth issues of latency. The best bandwidth available would be achieved when all of the Module Controllers are working simultaneously. Thus, for example, if data is required from all PUPE engines, rather than read from each PUPE channel one after

the other it would be best to read from channels 1, 16 and 31 simultaneously then move to 2,17,32 etc.

# 8. Compiling the TMS Client Library

The TMS client library is shipped in the **tms-lib-src-<VERSION>.tar.gz** tar archive. The source code has been built and tested on an x86 Fedora Core 6 system, an x86 Redhat 7.3 system, an FPGALinux PowerPC system and the LynxOS system as used at CERN. Its should be portable to other POSIX based system with minor modifications.

In order to compile the library it first needs to be configured. To configure the TMS source for build first modify the Makefile.config file to suit your requirements. The main parameters to modify include:

- BUILD_ENV – Defines the build environment. Options are:

  “Undefined” - normal build environment

  “CERN” - CERN's internal build environment
- BUILD - Defines which components to build. Options are:

  "FULL" - for full TMS system,

  "CLIENT" - for client code,

  "LIB" - for just development libraries.

- TARGET - Defines the target platform to build for. Current settings include:

  "el6" - for Centos6 or Redhat enterpise Linux 6

  "lynxos" - for LynxOS.

Once configured the TMS libraries can be built using the commands:

1. make depend
2. make

Note that the “make depend” command may not function in some build environments including the CERN environment.

Once built the programs/libraries can be installed on a system, if required, using the following command run as the superuser:

1. make install

There are some example client applications using the TMS API libraries in the tmsExamples directory.

# 9. Building the main TMS software

The TMS software source code is shipped in the **tms-full-src-<VERSION>.tar.gz** compressed tar archive. The code is designed to be built on a Linux Fedora Core 6 system. It consists of 3 main parts: The TMS main code, the Module controller OS and the FPGA packaging parts. The source tree from the tar archive can be located in any suitable directory on the system.

If you intend to build RPM packages, which is recommended, you will need access to the RPM_BUILD directory for this. By default this is set to /usr/src/redhat and thus requires root access. If you wish to build RPM's as a normal user, you can create your own local directory for the RPM build process and set the

RPM_BUILD parameter in Makefile.config files to point to this. You should then add "%_topdir <RPM_DIR>" to your personal ~/.rpmmacros file.

The TMS sofware is split into the following Modules:

- Tms – The main system software
- Tms-sys – System Controller configuration
- Tms-mcsys – Module Controller system
- Tms-fpga – FPGA firmware
- Tms-doc – system documentation

The tms-full-src-<VERSION>.tar.gz archive contains the full source code.

## 9.1. Building the main TMS software

The main TMS software consists of the TMS API libraries, the TMS server programs and the TMS client programs that run on the system controller and module controllers. The code is situated in the tms directory.

The following instructions describe how to build and package this code:

1. Run "make depend" to build the dependencies files.
2. If you update the code then you should first change the version number in the file Makefile.config.
3. Run "make" to build all of the libraries and programs.

A manual install of the software can be made by running the command "make install" as root on the system controller.

To create an RPM package run the command "make rpms". See the notes above if you wish to build the RPMS as a normal user.

The created RPM files can be copied into ../packages using the "make rpmsInstall" command.

## 9.2. Building the Module controller OS

The module controller's OS module, tms-mcsys, can be build on the TMS server. The kernel and driver modules of the system being used to build the package are used for the module controller's OS. So this build should be performed on a system controller.

To build the software:

1. Run "make config" to configure the build.
2. If you update the code then you should first change the version number in the file Makefile.config.
3. Run "make" to build all of the libraries and programs.

To create an RPM package run the command "make rpms". See the notes above if you wish to build the RPMS as a normal user.

## 9.3. Building a FPGA Firmware package

This section describes how to build an FPGA firmware RPM package. It assumes that a suitable FPGA bit file has already been created with the Xilinx FPGA tool set. The scripts to do this are in the tms-fpga directory.

1. Copy bit file to tms-fpga directory with appropriate file name containing the version number.
2. Symbolically link this bit file with tms-fpga.bit. (ln -sf tms-fpga-1.2.0.bit tms-fpga.bit)
3. Run “make rpm” to build package. See the notes above if you wish to build the RPMS as a normal user.
4. Run “make rpmInstall” to install the package in packages

# 10. Overview of BOAP RPC Operation

The BOAP (Beam Object Access protocol) system implements a simple, binary based, RPC system that runs across a TCP/IP link.. The system has a client/server architecture. A server will implement a set of service objects each of which have a number of possible methods (function calls). A client application can create a connection to these objects and call the method functions as required. An application can have a number of service objects and call methods on other applications service objects as required. The BOAP system can work in a single or multi-threaded environment.

The BOAP system employs an object name server that provides an object name to IP Address and socket lookup facility. When a server creates a new service object, its name, IP Address and socket number is registered with the BOAP name server. Clients can make use of the BOAP name server to connect to the required object using a simple ASCII string based name.

To ease the creation of service and client objects that implement the protocol, an interface definition language (IDL) compiler is provided. The BOAP interface definition language compiler takes a high level definition of the required objects and API and produces a documented set of service and client objects in 'C++' to implement that protocol. It can also produce other language interfaces to the same protocol.

At a low level the RPC system serialises the method/function call into a little endian binary sequence that is sent across a TCP/IP connection in an efficient manner. On a big endian system the data is automatically converted to big endian format. The system implements a set of basic primitive types and has the ability to implement higher level data structures, lists and multi-dimensional arrays.